Designing Virtual Knowledge Graphs

Diego Calvanese[1], **Benjamin Cogrel**[2], **Davide Lanti**

[1] Free University of Bozen-Bolzano, Italy
[2] Ontopic s.r.l

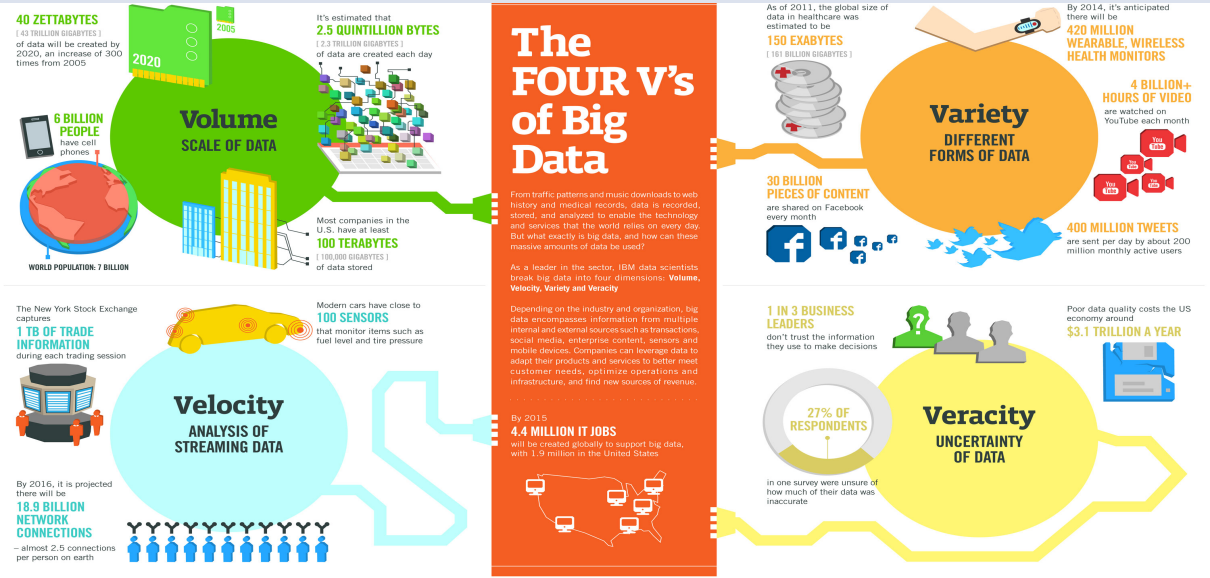
Funded by
the European Union

Tutorial at the
15th Int. Conf. on Formal Ontology in Information Systems (FOIS 2025)
8–12 September 2025 – Catania, Italy

Outline

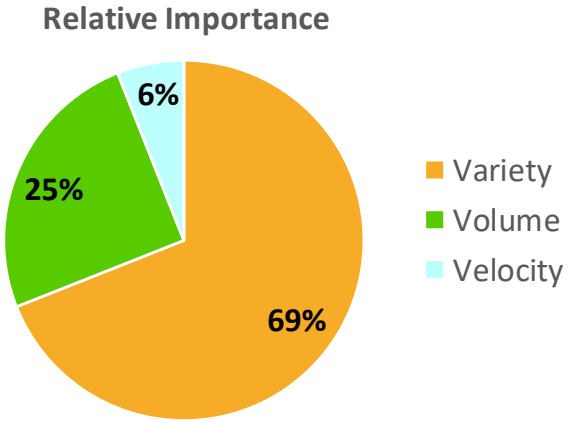
Outline

Challenges in the Big Data era



Variety, not volume, is driving Big Data initiatives

MIT Sloan Management Review (28 March 2016)



http://sloanreview.mit.edu/article/variety-not-volume-is-driving-big-data-initiatives/

How to deal with variety?

- In traditional data integration, the global (or mediated) schema is relational.
- In a relational schema, we have a limited ability of capturing domain knowledge.
- Domain knowledge can help in better relating to the information in the sources.
- By using domain knowledge, we can also automatically derive new data (or knowledge) from the one extracted from the sources.

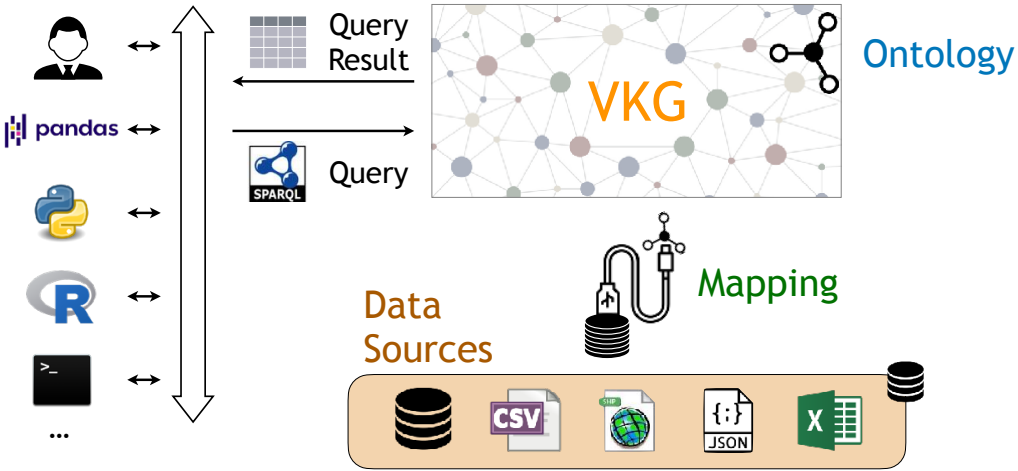
Virtual Knowledge Graphs for data access and integration (VKGs)

We combine three key ideas:

1. Use a global (or integrated) schema and **map the data sources to the global schema**.
2. Adopt a very flexible data model for the global schema
   $\rightsquigarrow$ **Knowledge Graph** whose vocabulary is expressed in an **ontology**.
3. Exploit **virtualization**, i.e., the KG is not materialized, but kept virtual.

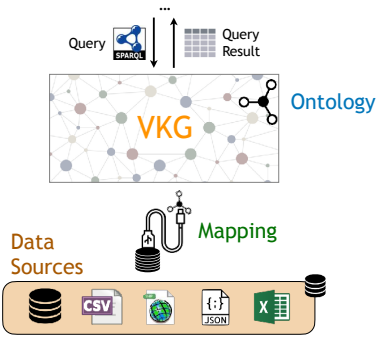
This gives rise to the **Virtual Knowledge Graph** (**VKG**) approach to data access / integration, also called **Ontology-based Data Access / Integration** (**OBDA**).
[Xiao et al. 2018, IJCAI]

Virtual Knowledge Graph (VKG) architecture



Why an ontology?

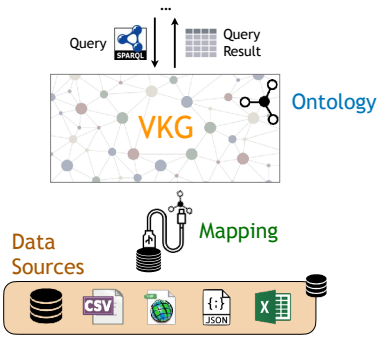
An ontology is a structured formal representation of concepts and their relationships that are relevant for the domain of interest.



- In the VKG setting, the ontology has a twofold purpose:
  - It defines a vocabulary of terms to denote classes and properties that are familiar to the user.
  - It extends the data in the sources with background knowledge about the domain of interest, and this knowledge is machine processable.
- One can make use of custom-built domain ontologies.
- In addition, one can rely on standard ontologies, which are available for many domains.

Why an ontology?

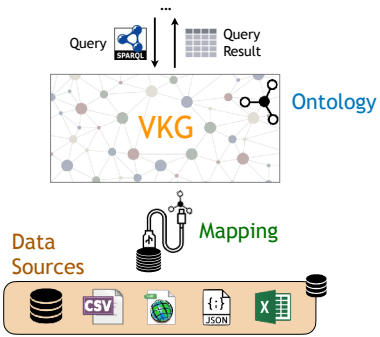
An ontology is a structured formal representation of concepts and their relationships that are relevant for the domain of interest.



- In the VKG setting, the ontology has a twofold purpose:
  - It defines a vocabulary of terms to denote classes and properties that are familiar to the user.
  - It extends the data in the sources with background knowledge about the domain of interest, and this knowledge is machine processable.
- One can make use of custom-built domain ontologies.
- In addition, one can rely on standard ontologies, which are available for many domains.

Why a Knowledge Graph for the global schema?

The traditional approach to data integration adopts a relational global schema.

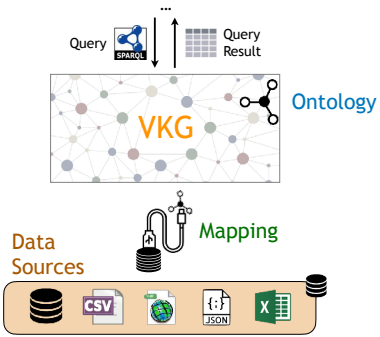


A Knowledge Graph, instead:

- Handles heterogeneous, incomplete data seamlessly.
- Easily integrates new sources without schema redesign.
- Provides context that boosts ML performance, and accommodates for inference.
- Enables transparent explanations via provenance and links.
- Supports FAIR principles for governance and re-use.

Why a Knowledge Graph for the global schema?

The traditional approach to data integration adopts a relational global schema.

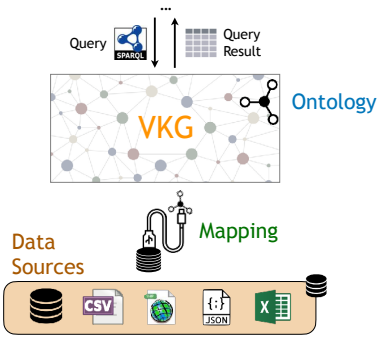


A Knowledge Graph, instead:

- Handles heterogeneous, incomplete data seamlessly.
- Easily integrates new sources without schema redesign.
- Provides context that boosts ML performance, and accommodates for inference.
- Enables transparent explanations via provenance and links.
- Supports FAIR principles for governance and re-use.

Why mappings?

The traditional approach to data integration relies on mediators, which are specified through complex code.

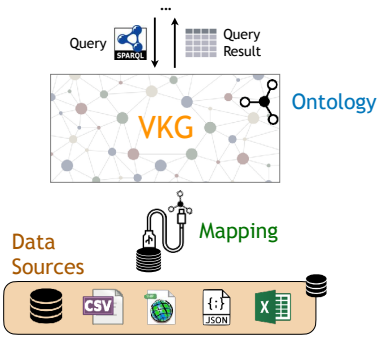


Mappings, instead:

- Provide a declarative specification, and not code.
- Are easier to understand, and hence to design and to maintain.
- Support an incremental approach to integration.
- Are machine processable, hence are used in query answering and for query optimization.

Why mappings?

The traditional approach to data integration relies on mediators, which are specified through complex code.

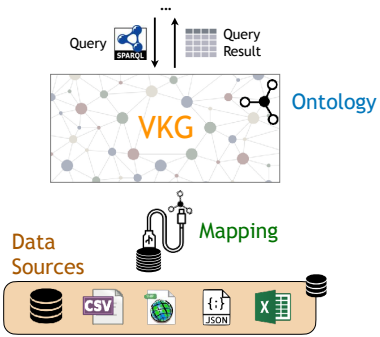


Mappings, instead:

- Provide a declarative specification, and not code.
- Are easier to understand, and hence to design and to maintain.
- Support an incremental approach to integration.
- Are machine processable, hence are used in query answering and for query optimization.

Why virtualization?

Materialized data integration relies on extract-transform-load (ETL) operations, to load data from the sources into an integrated data store / data warehouse / materialized KG.

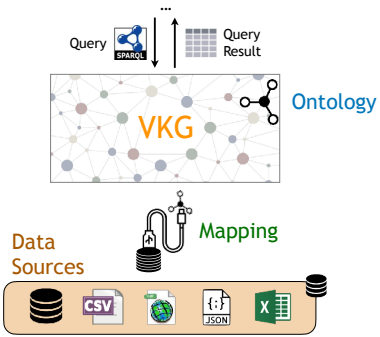


In the virtual approach, instead:

- The data stays in the sources and is only accessed at query time.
- No need to construct a large and potentially costly materialized data store and keep it up-to-date.
- Hence the data is always fresh wrt the latest updates at the sources.
- One can rely on the existing data infrastructure and expertise.
- There is better support for an incremental approach to integration.

Why virtualization?

Materialized data integration relies on extract-transform-load (ETL) operations, to load data from the sources into an integrated data store / data warehouse / materialized KG.



In the virtual approach, instead:

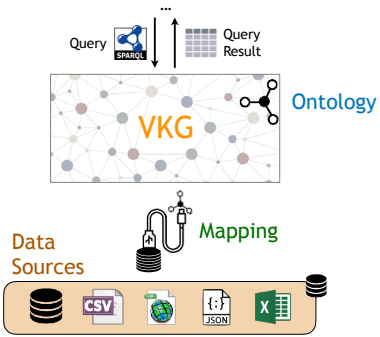
- The data stays in the sources and is only accessed at query time.
- No need to construct a large and potentially costly materialized data store and keep it up-to-date.
- Hence the data is always fresh wrt the latest updates at the sources.
- One can rely on the existing data infrastructure and expertise.
- There is better support for an incremental approach to integration.

Incomplete information

We are in a setting of **incomplete information**!!!

Incompleteness is introduced:

- by data sources, in general assumed to be incomplete;
- by domain constraints encoded in the ontology.

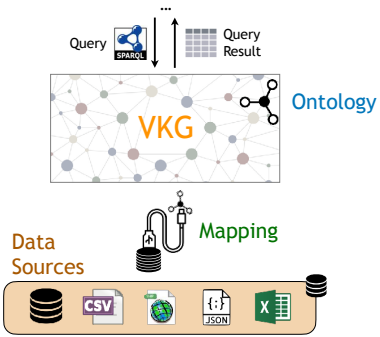


Incomplete information

We are in a setting of **incomplete information**!!!

Incompleteness is introduced:

- by data sources, in general assumed to be incomplete;
- by domain constraints encoded in the ontology.



Plus:

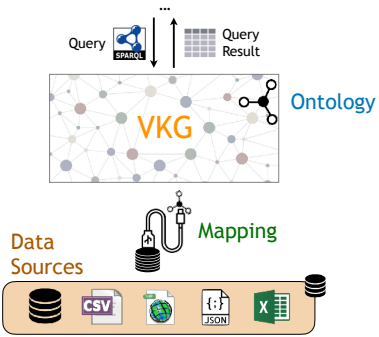
Ontologies are logical theories, and hence perfectly suited to deal with incomplete information!

Incomplete information

We are in a setting of **incomplete information**!!!

Incompleteness is introduced:

- by data sources, in general assumed to be incomplete;
- by domain constraints encoded in the ontology.



Plus:

Ontologies are logical theories, and hence perfectly suited to deal with incomplete information!

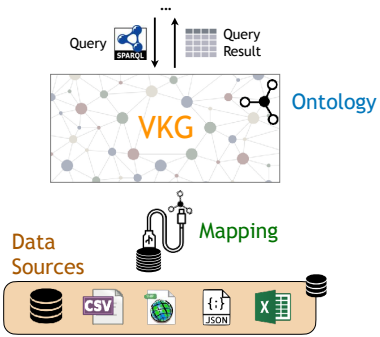
Minus:

Query answering amounts to **logical inference**, and hence is significantly more challenging.

Components of the VKG framework

We consider now the main components that make up the VKG framework, and the languages used to specify them.

In defining such languages, we need to consider the **tradeoff between expressive power and efficiency**, where the key point is efficiency with respect to the data.



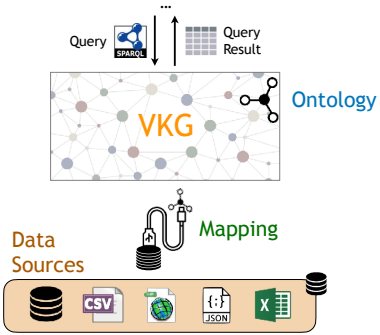
The W3C has standardized languages that are suitable for VKGs:

1. Knowledge graph: expressed in **RDF** [W3C Rec. 2014] (v1.1)
2. Ontology $\mathcal{O}$: expressed in **OWL 2 QL** [W3C Rec. 2012]
3. Mapping $\mathcal{M}$: expressed in **R2RML** [W3C Rec. 2012]
4. Query: expressed in **SPARQL** [W3C Rec. 2013] (v1.1)

Components of the VKG framework

We consider now the main components that make up the VKG framework, and the languages used to specify them.

In defining such languages, we need to consider the **tradeoff between expressive power and efficiency**, where the key point is efficiency with respect to the data.



The W3C has standardized languages that are suitable for VKGs:

1. Knowledge graph: expressed in **RDF** [W3C Rec. 2014] (v1.1)
2. Ontology $\mathcal{O}$: expressed in **OWL 2 QL** [W3C Rec. 2012]
3. Mapping $\mathcal{M}$: expressed in **R2RML** [W3C Rec. 2012]
4. Query: expressed in **SPARQL** [W3C Rec. 2013] (v1.1)

Outline

Outline

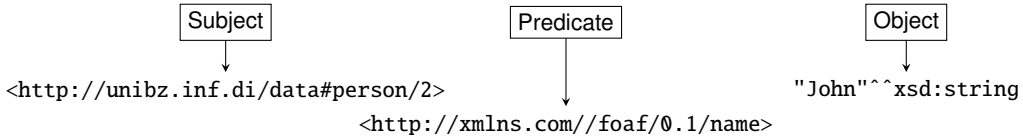
Resource Description Framework (RDF)

- RDF is a language standardized by the W3C for representing information [W3C Rec. 2004] (v1.0) and [W3C Rec. 2014] (v1.1).
- RDF is a **graph-based data model**, where information is represented as (labeled) nodes connected by (labeled) edges.
- Nodes have three different forms:
  - literal: denotes a constant value, with an associated datatype;
  - IRI (for *internationalized resource identifier*): denotes a resource (i.e., an object), for which the IRI acts as an identifier;
  - blank node: represents an anonymous object.
- An IRI might also denote a **property**, connecting an object to a literal, or connecting two objects.

See also `https://www.w3.org/TR/rdf11-concepts/` for details.

RDF triples

RDF provides a description of the domain of interest in terms of **triples**:



Triple elements: resources denoted by global identifiers (IRIs)

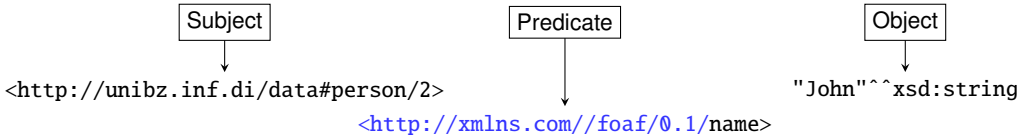
1. Subject: IRI of the described resource
2. Predicate: IRI of the property
3. Object: attribute value or IRI of another resource

Prefixes: useful abbreviations and/or references to external information

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

RDF triples

RDF provides a description of the domain of interest in terms of **triples**:



Triple elements: resources denoted by global identifiers (IRIs)

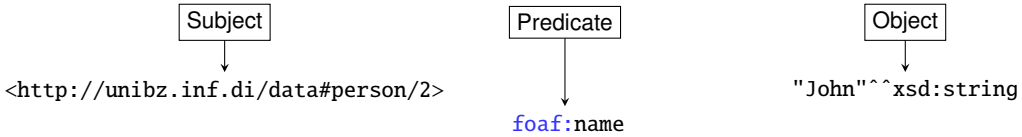
1. Subject: IRI of the described resource
2. Predicate: IRI of the property
3. Object: attribute value or IRI of another resource

Prefixes: useful abbreviations and/or references to external information

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

RDF triples

RDF provides a description of the domain of interest in terms of **triples**:



Triple elements: resources denoted by **global identifiers** (IRIs)

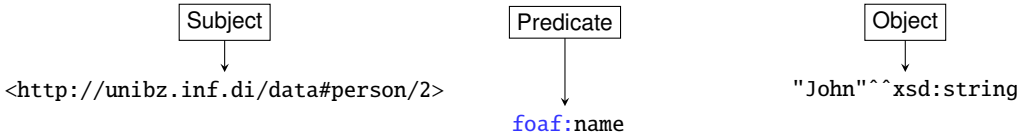
1. Subject: IRI of the described resource
2. Predicate: IRI of the property
3. Object: attribute value or IRI of another resource

Prefixes: useful abbreviations and/or references to external information

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

RDF triples

RDF provides a description of the domain of interest in terms of **triples**:



Triple elements: resources denoted by **global identifiers** (IRIs)

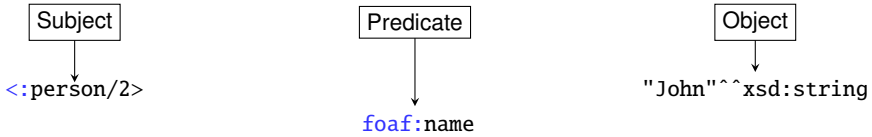
1. Subject: IRI of the described resource
2. Predicate: IRI of the property
3. Object: attribute value or IRI of another resource

Prefixes: useful abbreviations and/or references to external information

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix : <http://unibz.inf.di/data#>
```

RDF triples

RDF provides a description of the domain of interest in terms of **triples**:



Triple elements: resources denoted by **global identifiers** (IRIs)

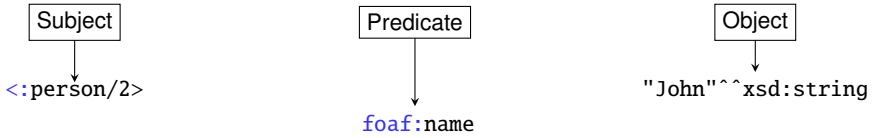
1. Subject: IRI of the described resource
2. Predicate: IRI of the property
3. Object: attribute value or IRI of another resource

Prefixes: useful abbreviations and/or references to external information

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix : <http://unibz.inf.di/data#>
```

RDF triples

RDF provides a description of the domain of interest in terms of **triples**:



Triple elements: resources denoted by **global identifiers** (IRIs)

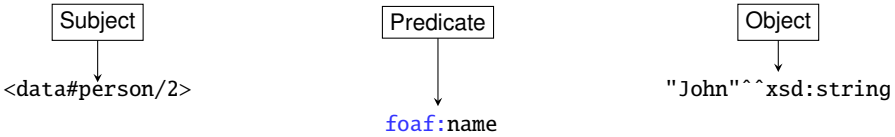
1. Subject: IRI of the described resource
2. Predicate: IRI of the property
3. Object: attribute value or IRI of another resource

Prefixes: useful abbreviations and/or references to external information

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix : <http://unibz.inf.di/data#>
@base <http://unibz.inf.di/>
```

RDF triples

RDF provides a description of the domain of interest in terms of **triples**:



Triple elements: resources denoted by **global identifiers** (IRIs)

1. Subject: IRI of the described resource
2. Predicate: IRI of the property
3. Object: attribute value or IRI of another resource

Prefixes: useful abbreviations and/or references to external information

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix : <http://unibz.inf.di/data#>
@base <http://unibz.inf.di/>
```

RDF – Examples

Class membership:

RDF triple	`<uni2/p/25> rdf:type :Professor`
Fact	Professor(*uni2/p/25*)

Note: This is typically abbreviated as

RDF triple	`<uni2/p/25> a :Professor`

Data property of an individual:

RDF triple	`<uni2/p/25> :lastName "Artale"`
Fact	lastName(*uni2/p/25*, "Artale")

Object property of an individual:

RDF triple	`<uni2/p/25> :teaches <uni2/c/7>`
Fact	teaches(*uni2/p/25*, *uni2/c/7*)

RDF – Examples

Class membership:

RDF triple	`<uni2/p/25> rdf:type :Professor`
Fact	Professor(*uni2/p/25*)

Note: This is typically abbreviated as

RDF triple	`<uni2/p/25> a :Professor`

Data property of an individual:

RDF triple	`<uni2/p/25> :lastName "Artale"`
Fact	lastName(*uni2/p/25*, "Artale")

Object property of an individual:

RDF triple	`<uni2/p/25> :teaches <uni2/c/7>`
Fact	teaches(*uni2/p/25*, *uni2/c/7*)

RDF – Examples

Class membership:

RDF triple	`<uni2/p/25> rdf:type :Professor`
Fact	Professor(*uni2/p/25*)

Note: This is typically abbreviated as

RDF triple	`<uni2/p/25> a :Professor`

Data property of an individual:

RDF triple	`<uni2/p/25> :lastName "Artale"`
Fact	lastName(*uni2/p/25*, "Artale")

Object property of an individual:

RDF triple	`<uni2/p/25> :teaches <uni2/c/7>`
Fact	teaches(*uni2/p/25*, *uni2/c/7*)

RDF – Examples

Class membership:

RDF triple	`<uni2/p/25> rdf:type :Professor`
Fact	Professor(*uni2/p/25*)

Note: This is typically abbreviated as

RDF triple	`<uni2/p/25> a :Professor`

Data property of an individual:

RDF triple	`<uni2/p/25> :lastName "Artale"`
Fact	lastName(*uni2/p/25*, "Artale")

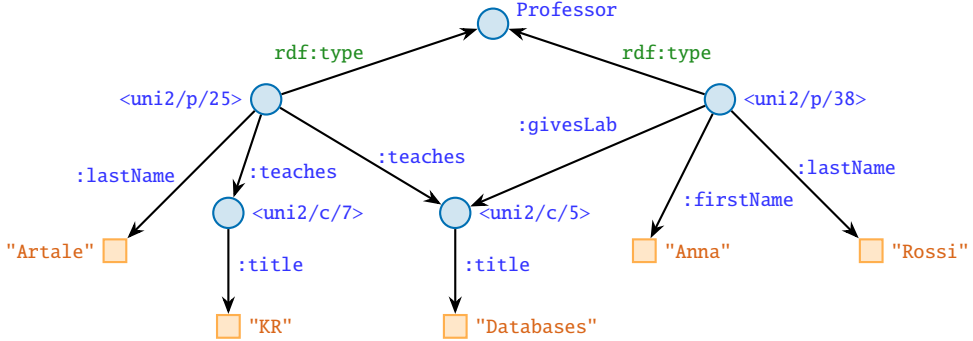
Object property of an individual:

RDF triple	`<uni2/p/25> :teaches <uni2/c/7>`
Fact	teaches(*uni2/p/25*, *uni2/c/7*)

RDF graph – Example

```
<uni2/p/25> rdf:type :Professor
<uni2/p/25> foaf:lastName "Artale"
<uni2/p/25> :teaches <uni2/c/5>
...
```

We can represent such a set of facts graphically:



RDF datatypes

- Datatypes are used with RDF literals to represent values such as strings, numbers, and dates.
- Each datatype is itself denoted by an IRI. E.g., the XML Schema built-in datatypes have IRIs of the form `http://www.w3.org/2001/XMLSchema#xxx`
- Each datatype associates to elements in a **lexical space** (i.e., unicode strings) elements from a **value space**.
  Example:
  - datatype: `xsd:boolean`
  - lexical space: { “`true`”, “`false`”, “`1`”, “`0`” }
  - value space: $\{true, false\}$
- To explicitly associate a datatype to a literal, we use the notation *literal*`^^`*datatype*.
  Example: `12.5^^xsd:double`, `1^^xsd:integer`

XML Schema built-in datatypes (recommended)

	Datatype	Value space (informative)
Core types	xsd:string	Character strings
	xsd:boolean	true, false
	xsd:decimal	Arbitrary-precision decimal numbers
	xsd:integer	Arbitrary-size integer numbers
IEEE floating-point numbers	xsd:float	32-bit floating point numbers incl. ±Inf, ±0, NaN
	xsd:double	64-bit floating point numbers incl. ±Inf, ±0, NaN
Time and date	xsd:date	Dates (yyyy-mm-dd) with or without timezone
	xsd:time	Times (hh:mm:ss.sss...) with or without timezone
	xsd:datetime	Date and time with or without timezone
Limited-range integer numbers	xsd:byte	8 bit integers (-128, ..., +127)
	xsd:short	16 bit integers
	xsd:int	32 bit integers
	xsd:long	64 bit integers
	xsd:unsignedByte	8 bit non-negative integers (0, ..., 255)
	xsd:unsignedShort	16 bit non-negative integers
	...	

Additional RDF features

RDF has additional features that we do not cover here:

- blank nodes
- named graphs

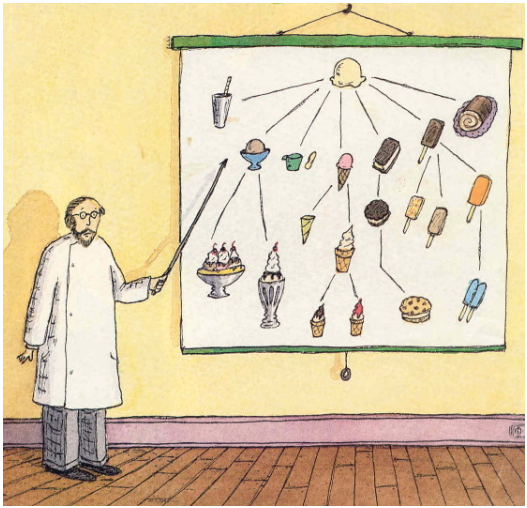
Outline

What is an ontology?

- An ontology conceptualizes a domain of interest in terms of **concepts/classes**, (binary) **relations**, and their **properties**.
- It typically organizes the concepts in a hierarchical structure.
- Ontologies are often represented as graphs.
- However, an ontology is actually a **logical theory**, expressed in a suitable fragment of first-order logic, or better, in **description logics**.

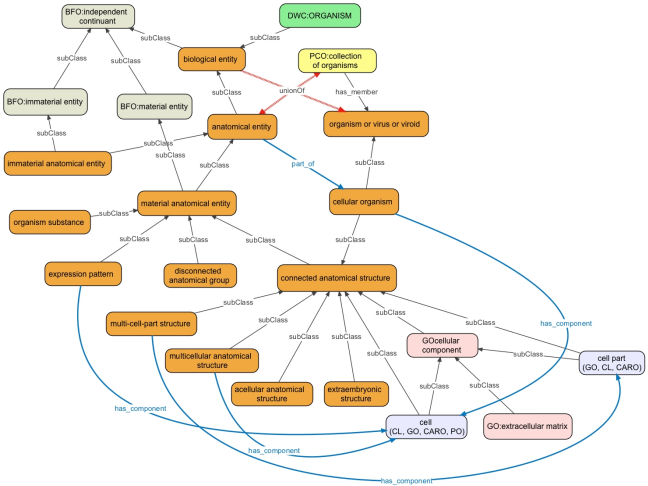
What is an ontology?

- An ontology conceptualizes a domain of interest in terms of **concepts/classes**, (binary) **relations**, and their **properties**.
- It typically organizes the concepts in a hierarchical structure.
- Ontologies are often represented as graphs.
- However, an ontology is actually a **logical theory**, expressed in a suitable fragment of first-order logic, or better, in **description logics**.



What is an ontology?

- An ontology conceptualizes a domain of interest in terms of **concepts/classes**, (binary) **relations**, and their **properties**.
- It typically organizes the concepts in a hierarchical structure.
- Ontologies are often represented as graphs.
- However, an ontology is actually a **logical theory**, expressed in a suitable fragment of first-order logic, or better, in **description logics**.



What is an ontology?

- An ontology conceptualizes a domain of interest in terms of **concepts/classes**, (binary) **relations**, and their **properties**.
- It typically organizes the concepts in a hierarchical structure.
- Ontologies are often represented as graphs.
- However, an ontology is actually a **logical theory**, expressed in a suitable fragment of first-order logic, or better, in **description logics**.

$\forall x.\, \mathsf{Actor}(x) \rightarrow \mathsf{Staff}(x)$
$\forall x.\, \mathsf{SeriesActor}(x) \rightarrow \mathsf{Actor}(x)$
$\forall x.\, \mathsf{MovieActor}(x) \rightarrow \mathsf{Actor}(x)$
$\forall x.\, \mathsf{SeriesActor}(x) \rightarrow \neg\mathsf{MovieActor}(x)$

$\forall x.\, \mathsf{Staff}(x) \rightarrow \exists y.\, \mathit{ssn}(x, y)$
$\forall y.\, \exists x.\, \mathit{ssn}(x, y) \rightarrow \mathtt{xsd{:}int}(y)$
$\forall x, y, y'.\, \mathit{ssn}(x, y) \wedge \mathit{ssn}(x, y') \rightarrow y = y'$

$\forall x.\, \exists y.\, \mathsf{actsIn}(x, y) \rightarrow \mathsf{MovieActor}(x)$
$\forall y.\, \exists x.\, \mathsf{actsIn}(x, y) \rightarrow \mathsf{Movie}(y)$
$\forall x.\, \mathsf{MovieActor}(x) \rightarrow \exists y.\, \mathsf{actsIn}(x, y)$
$\forall x.\, \mathsf{Movie}(x) \rightarrow \exists y.\, \mathsf{actsIn}(y, x)$
$\forall x, y.\, \mathsf{actsIn}(x, y) \rightarrow \mathsf{playsIn}(x, y)$
$\ldots$

What is an ontology?

- An ontology conceptualizes a domain of interest in terms of **concepts/classes**, (binary) **relations**, and their **properties**.
- It typically organizes the concepts in a hierarchical structure.
- Ontologies are often represented as graphs.
- However, an ontology is actually a **logical theory**, expressed in a suitable fragment of first-order logic, or better, in **description logics**.

$$
\begin{aligned}
\text{Actor} &\sqsubseteq \text{Staff} \\
\text{SeriesActor} &\sqsubseteq \text{Actor} \\
\text{MovieActor} &\sqsubseteq \text{Actor} \\
\text{SeriesActor} &\sqsubseteq \neg\text{MovieActor}
\end{aligned}
$$

$$
\begin{aligned}
\text{Staff} &\sqsubseteq \exists \mathit{ssn} \\
\exists \mathit{ssn}^- &\sqsubseteq \texttt{xsd:int} \\
&(\textbf{funct}\ \mathit{ssn})
\end{aligned}
$$

$$
\begin{aligned}
\exists \text{actsIn} &\sqsubseteq \text{MovieActor} \\
\exists \text{actsIn}^- &\sqsubseteq \text{Movie} \\
\text{MovieActor} &\sqsubseteq \exists \text{actsIn} \\
\text{Movie} &\sqsubseteq \exists \text{actsIn}^- \\
\text{actsIn} &\sqsubseteq \text{playsIn} \\
&\ldots
\end{aligned}
$$

The OWL 2 QL ontology language

- **OWL 2 QL** is one of the three standard profiles of OWL 2. [W3C Rec. 2012]
- Is derived from the *DL-Lite*$_{\mathcal{R}}$ description logic (DL) [Baader et al. 2003] of the *DL-Lite*-family.
- Is considered a lightweight ontology language:
  - controlled expressive power
  - efficient inference
- Optimized for accessing large amounts of data (i.e., for data complexity):
  - Queries over the *data modulo the ontology* can be rewritten into SQL queries over the underlying relational database (First-order rewritability of query answering).
  - Consistency of ontology and data can also be checked by executing SQL queries (i.e., it is also first-order rewritable).

Classes and properties in OWL 2 QL

All ontology languages based on OWL 2 (and hence also OWL 2 QL), provide three types of elements to construct an ontology:

- **Classes** (also called concepts), which allow one to structure the domain of interest, by grouping in a class objects with common properties.
  Examples: Movie, Staff, Actor, SeriesActor, ...
- **Data properties** (also called attributes), which are binary relations that relate objects to values (or literals, in RDF terminology).
  Examples:
  - title, associating a string to a Movie;
  - ssn, associating an integer to a Person.
- **Object properties** (also called roles), which are binary relations between objects.
  Examples:
  - actsIn, relating a MovieActor to a Movie;
  - worksFor, relating an Employee to a Project.

In the following, to depict an OWL 2 QL ontology, we make use of a **graphical notation** inspired by the one for UML class diagrams.

Classes and properties in OWL 2 QL

All ontology languages based on OWL 2 (and hence also OWL 2 QL), provide three types of elements to construct an ontology:

- **Classes** (also called concepts), which allow one to structure the domain of interest, by grouping in a class objects with common properties.
  Examples: Movie, Staff, Actor, SeriesActor, ...
- **Data properties** (also called attributes), which are binary relations that relate objects to values (or literals, in RDF terminology).
  Examples:
  - title, associating a string to a Movie;
  - ssn, associating an integer to a Person.
- **Object properties** (also called roles), which are binary relations between objects.
  Examples:
  - actsIn, relating a MovieActor to a Movie;
  - worksFor, relating an Employee to a Project.

In the following, to depict an OWL 2 QL ontology, we make use of a **graphical notation** inspired by the one for UML class diagrams.

Classes and properties in OWL 2 QL

All ontology languages based on OWL 2 (and hence also OWL 2 QL), provide three types of elements to construct an ontology:

- **Classes** (also called concepts), which allow one to structure the domain of interest, by grouping in a class objects with common properties.
  Examples: Movie, Staff, Actor, SeriesActor, ...
- **Data properties** (also called attributes), which are binary relations that relate objects to values (or literals, in RDF terminology).
  Examples:
  - title, associating a string to a Movie;
  - ssn, associating an integer to a Person.
- **Object properties** (also called roles), which are binary relations between objects.
  Examples:
  - actsIn, relating a MovieActor to a Movie;
  - worksFor, relating an Employee to a Project.

In the following, to depict an OWL 2 QL ontology, we make use of a **graphical notation** inspired by the one for UML class diagrams.

Classes and properties in OWL 2 QL

All ontology languages based on OWL 2 (and hence also OWL 2 QL), provide three types of elements to construct an ontology:

- **Classes** (also called concepts), which allow one to structure the domain of interest, by grouping in a class objects with common properties.
  Examples: Movie, Staff, Actor, SeriesActor, ...
- **Data properties** (also called attributes), which are binary relations that relate objects to values (or literals, in RDF terminology).
  Examples:
  - title, associating a string to a Movie;
  - ssn, associating an integer to a Person.
- **Object properties** (also called roles), which are binary relations between objects.
  Examples:
  - actsIn, relating a MovieActor to a Movie;
  - worksFor, relating an Employee to a Project.

In the following, to depict an OWL 2 QL ontology, we make use of a **graphical notation** inspired by the one for UML class diagrams.

OWL 2 QL knowledge bases

An **OWL 2 QL knowledge base (KB)** $\mathcal{K} = (\mathcal{O}, \mathcal{G})$ consists of two parts:

An **ontology** $\mathcal{O}$ modeling the schema level information.

- Contains the declarations of the classes, data properties, and object properties of the ontology. This constitutes the **vocabulary** with which we can then query the ontology.
- Contains the **axioms** that capture the **domain knowledge**.
- These axioms express the conditions that must hold for the classes and properties in the ontology.

An **RDF graph** $\mathcal{G}$, modeling the extensional level information (i.e., facts).

The RDF graph $\mathcal{G}$ consists of triples that express membership assertions of the following forms:

- An individual `<a>` belongs to a class `:C`: `<a> rdf:type :C .`
- A pair individual `<a>` and literal `<l>` belongs to a data property `:A`: `<a> :A <l> .`
- A pair of individuals `<a1>`, `<a2>` belongs to an object property `:P`: `<a1> :P <a2> .`

Note: As we will see later, in the VKG setting, the RDF graph of a KB is not given explicitly, but is (usually) defined implicitly through the database(s) and the mappings.

OWL 2 QL knowledge bases

An **OWL 2 QL knowledge base (KB)** $\mathcal{K} = (\mathcal{O}, \mathcal{G})$ consists of two parts:

An **ontology** $\mathcal{O}$ modeling the schema level information.

- Contains the declarations of the classes, data properties, and object properties of the ontology. This constitutes the **vocabulary** with which we can then query the ontology.
- Contains the **axioms** that capture the **domain knowledge**.
- These axioms express the conditions that must hold for the classes and properties in the ontology.

An **RDF graph** $\mathcal{G}$, modeling the extensional level information (i.e., facts).

The RDF graph $\mathcal{G}$ consists of triples that express membership assertions of the following forms:

- An individual `<a>` belongs to a class `:C`: `<a> rdf:type :C .`
- A pair individual `<a>` and literal `<l>` belongs to a data property `:A`: `<a> :A <l> .`
- A pair of individuals `<a1>`, `<a2>` belongs to an object property `:P`: `<a1> :P <a2> .`

Note: As we will see later, in the VKG setting, the RDF graph of a KB is not given explicitly, but is (usually) defined implicitly through the database(s) and the mappings.

Declaration of classes

Declaration of a class C

```
C rdf:type owl:Class .
```

`owl:Class` is a predefined class in OWL 2, whose instances are all the classes of an ontology.

When a class has no data properties (or the data properties are not of interest) we represent the class simply as a rounded rectangle that contains the class name.

Example:

Declaration of classes and of data properties

Declaration of a class C

```
C rdf:type owl:Class .
```

`owl:Class` is a predefined class in OWL 2, whose instances are all the classes of an ontology.

When a class has no data properties (or the data properties are not of interest) we represent the class simply as a rounded rectangle that contains the class name.

Example:

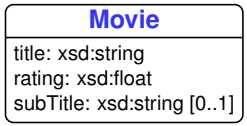
Declaration of a data property A

```
A rdf:type owl:DatatypeProperty
```

`owl:DatatypeProperty` is a predefined class in OWL 2, whose instances are all the data properties of an ontology.

The data properties for a class are typically depicted together with the class itself.
In that case, we split the rectangle in two, and we specify the data properties of the class in the bottom part.

Example:



Declaration of object properties

Declaration of an object property P

```
P rdf:type owl:ObjectProperty .
```

`owl:ObjectProperty` is a predefined class in OWL 2, whose instances are all object properties of an ontology.

In the graphical notation, we represent an object property by an arrow that connects two classes and that is labeled with the name of the object property.

Example:

The arrow might additionally be labeled with cardinalities. These are pairs of numbers, representing the minimum and maximum number of connections that an individual might have for the property.

Note: Each data property, and each direct and inverse object property has a **cardinality**.
In the graphical notation, when the cardinalities are missing, we assume the following defaults:

- [1..1] for a data property;
- 0..* (i.e., no constraint) for an object property;
- 0..* (i.e., no constraint) for the inverse of an object property.

Declaration of object properties

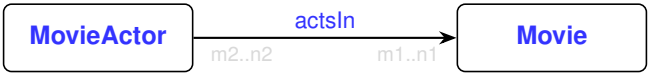
Declaration of an object property P

```
P rdf:type owl:ObjectProperty .
```

`owl:ObjectProperty` is a predefined class in OWL 2, whose instances are all object properties of an ontology.

In the graphical notation, we represent an object property by an arrow that connects two classes and that is labeled with the name of the object property.

Example:



The arrow might additionally be labeled with cardinalities. These are pairs of numbers, representing the minimum and maximum number of connections that an individual might have for the property.

Note: Each data property, and each direct and inverse object property has a **cardinality**.
In the graphical notation, when the cardinalities are missing, we assume the following defaults:

- [1..1] for a data property;
- 0..* (i.e., no constraint) for an object property;
- 0..* (i.e., no constraint) for the inverse of an object property.

Declaration of object properties

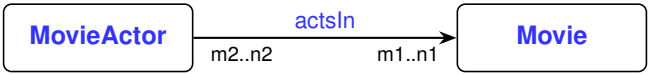
Declaration of an object property P

```
P rdf:type owl:ObjectProperty .
```

`owl:ObjectProperty` is a predefined class in OWL 2, whose instances are all object properties of an ontology.

In the graphical notation, we represent an object property by an arrow that connects two classes and that is labeled with the name of the object property.

Example:



The arrow might additionally be labeled with cardinalities. These are pairs of numbers, representing the minimum and maximum number of connections that an individual might have for the property.

Note: Each data property, and each direct and inverse object property has a **cardinality**.
In the graphical notation, when the cardinalities are missing, we assume the following defaults:

- [1..1] for a data property;
- 0..* (i.e., no constraint) for an object property;
- 0..* (i.e., no constraint) for the inverse of an object property.

Declaration of object properties

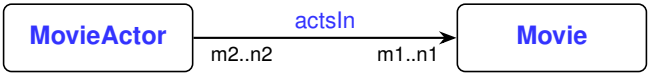
Declaration of an object property P

```
P rdf:type owl:ObjectProperty .
```

`owl:ObjectProperty` is a predefined class in OWL 2, whose instances are all object properties of an ontology.

In the graphical notation, we represent an object property by an arrow that connects two classes and that is labeled with the name of the object property.

Example:



The arrow might additionally be labeled with cardinalities. These are pairs of numbers, representing the minimum and maximum number of connections that an individual might have for the property.

Note: Each data property, and each direct and inverse object property has a **cardinality**.
In the graphical notation, when the cardinalities are missing, we assume the following defaults:

- [1..1] for a data property;
- 0..* (i.e., no constraint) for an object property;
- 0..* (i.e., no constraint) for the inverse of an object property.

Semantics of OWL 2 QL KBs

An **interpretation** $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of a KB $\mathcal{K} = (\mathcal{O}, \mathcal{G})$ consists of:

- a nonempty set $\Delta^{\mathcal{I}}$, called the interpretation domain (of $\mathcal{I}$), and
- an interpretation function $\cdot^{\mathcal{I}}$, which maps
  - each constant (i.e., individual or literal) c to itself, i.e., $c^{\mathcal{I}} = c$; **(standard name assumption)**
  - each class name C to a subset $C^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$
  - each (object or data) property name P to a subset $P^{\mathcal{I}}$ of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
- The interpretation function is then extended to cover the OWL 2 QL constructs:

$$(P^-)^{\mathcal{I}} = \{(y, x) \mid (x, y) \in P^{\mathcal{I}}\} \qquad \exists R^{\mathcal{I}} = \{x \mid \text{there is some } y \text{ such that } (x, y) \in R^{\mathcal{I}}\}$$
$$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$$

Model

An interpretation $\mathcal{I}$ is a **model** of a KB $\mathcal{K} = (\mathcal{O}, \mathcal{G})$, denoted as $\mathcal{I} \models \mathcal{K}$, if it **satisfies** all axioms in $\mathcal{O}$ and assertions in $\mathcal{G}$.

In the next slides, we specify what these axioms/assertions are, as well as the satisfaction conditions.

Axioms in an OWL 2 QL ontology

We discuss now the various types of axioms that can be used in an OWL 2 QL ontology to capture domain knowledge.

Notes:

- Some of these axioms are part of the RDF Schema (RDFS) language, which is a fragment of OWL 2 QL, while others go beyond what can be expressed in RDFS.
- In the following, when we talk about a 'constant' we mean either an individual a (denoted by an IRI) or a literal ℓ.
- On the slides, for the assertions that make up the RDF graph, instead of the triple notation we also make use of a more compact (abstract) notation:

$C(a)$	for	`<a> rdf:type :C .`	(membership assertion in a class)
$A(a, \ell)$	for	`<a> :A <l> .`	(membership assertion in a data property)
$P(a_1, a_2)$	for	`<a1> :P <a2> .`	(membership assertion in an object property)

Syntax and semantics of OWL 2 QL KBs

Axiom type	OWL Syntax	DL Syntax	Semantics
Membership (class)	`<a> rdf:type <C>`	$C(a)$	$a \in C^{\mathcal{I}}$
Membership (data property)	`<a> :A <l>`	$A(a, \ell)$	$(a, \ell) \in A^{\mathcal{I}}$
Membership (object property)	`<a1> :P <a2>`	$P(a_1, a_2)$	$(a_1, a_2) \in P^{\mathcal{I}}$
Subclass assertion	`C1 rdfs:subClassOf C2`	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Class disjointness	`C1 owl:disjointWith C2`	$C_1 \sqsubseteq \neg C_2$	$C_1^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} - C_2^{\mathcal{I}}$
Property disjointness	`P1 owl:propertyDisjointWith P2`	$P_1 \sqsubseteq \neg P_2$	$P_1^{\mathcal{I}} \subseteq (\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) - P_2^{\mathcal{I}}$
Domain of a property	`P rdfs:domain C1`	$\exists P \sqsubseteq C_1$	$\{x \mid \exists y.(x, y) \in P^{\mathcal{I}}\} \subseteq C_1^{\mathcal{I}}$
Range of a property	`P rdfs:range C2`	$\exists P^- \sqsubseteq C_2$	$\{y \mid \exists x.(x, y) \in P^{\mathcal{I}}\} \subseteq C_2^{\mathcal{I}}$
Mandatory participation	using `owl:someValuesFrom`	$C \sqsubseteq \exists R$	$C^{\mathcal{I}} \subseteq \exists R^{\mathcal{I}}$
Subproperty assertion	`P1 rdfs:subPropertyOf R2`	$P_1 \sqsubseteq R_2$	$P_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
Inverse property	`P2 owl:inverseOf P1`	$P_1 \equiv P_2^-$	$P_1^{\mathcal{I}} = \{(y, x) \mid (x, y) \in P_2^{\mathcal{I}}\}$

- We have used R to denote either an object property P or the inverse P^- of an object property.
- We have listed the axioms involving object properties, but OWL 2 QL allows for analogous axioms involving data properties.

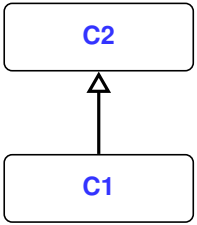
RDF Schema – Class hierarchy

Class hierarchy

```
C1 rdfs:subClassOf C2 .
```

$C_1 \sqsubseteq C_2$

When class C_1 is declared to be a **sub-class of** class C_2, then every object that is an instance of C_1 is also an instance of C_2.



Example: `:MovieActor rdfs:subClassOf :Actor .`

Inference: `<person/2> rdf:type :MovieActor .`
$\Longrightarrow$ `<person/2> rdf:type :Actor .`

In DL notation: $\mathit{MovieActor} \sqsubseteq \mathit{Actor}$

$\mathit{MovieActor}(\mathit{person/2}) \Longrightarrow \mathit{Actor}(\mathit{person/2})$

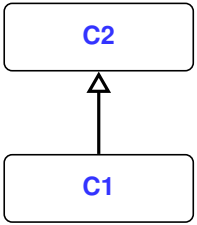
RDF Schema – Class hierarchy

Class hierarchy

```
C1 rdfs:subClassOf C2 .
```

$C_1 \sqsubseteq C_2$

When class C_1 is declared to be a **sub-class of** class C_2, then every object that is an instance of C_1 is also an instance of C_2.



Example: `:MovieActor rdfs:subClassOf :Actor .`

Inference: `<person/2> rdf:type :MovieActor .`

$\Longrightarrow$ `<person/2> rdf:type :Actor .`

In DL notation: $\mathit{MovieActor} \sqsubseteq \mathit{Actor}$

$\mathit{MovieActor}(\mathit{person/2}) \Longrightarrow \mathit{Actor}(\mathit{person/2})$

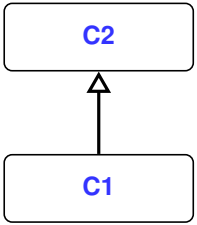
RDF Schema – Class hierarchy

Class hierarchy

`C1 rdfs:subClassOf C2 .`

$C_1 \sqsubseteq C_2$

When class C_1 is declared to be a **sub-class of** class C_2, then every object that is an instance of C_1 is also an instance of C_2.



Example: `:MovieActor rdfs:subClassOf :Actor .`

Inference: `<person/2> rdf:type :MovieActor .`

$\Longrightarrow$ `<person/2> rdf:type :Actor .`

In DL notation: MovieActor $\sqsubseteq$ Actor

MovieActor(*person/2*) $\Longrightarrow$ Actor(*person/2*)

Syntax and semantics of OWL 2 QL KBs

Axiom type	OWL Syntax	DL Syntax	Semantics
Membership (class)	`<a> rdf:type <C>`	$C(a)$	$a \in C^{\mathcal{I}}$
Membership (data property)	`<a> :A <l>`	$A(a, \ell)$	$(a, \ell) \in A^{\mathcal{I}}$
Membership (object property)	`<a1> :P <a2>`	$P(a_1, a_2)$	$(a_1, a_2) \in P^{\mathcal{I}}$
Subclass assertion	`C1 rdfs:subClassOf C2`	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Class disjointness	`C1 owl:disjointWith C2`	$C_1 \sqsubseteq \neg C_2$	$C_1^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} - C_2^{\mathcal{I}}$
Property disjointness	`P1 owl:propertyDisjointWith P2`	$P_1 \sqsubseteq \neg P_2$	$P_1^{\mathcal{I}} \subseteq (\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) - P_2^{\mathcal{I}}$
Domain of a property	`P rdfs:domain C1`	$\exists P \sqsubseteq C_1$	$\{x \mid \exists y.(x, y) \in P^{\mathcal{I}}\} \subseteq C_1^{\mathcal{I}}$
Range of a property	`P rdfs:range C2`	$\exists P^- \sqsubseteq C_2$	$\{y \mid \exists x.(x, y) \in P^{\mathcal{I}}\} \subseteq C_2^{\mathcal{I}}$
Mandatory participation	using `owl:someValuesFrom`	$C \sqsubseteq \exists R$	$C^{\mathcal{I}} \subseteq \exists R^{\mathcal{I}}$
Subproperty assertion	`P1 rdfs:subPropertyOf R2`	$P_1 \sqsubseteq R_2$	$P_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
Inverse property	`P2 owl:inverseOf P1`	$P_1 \equiv P_2^-$	$P_1^{\mathcal{I}} = \{(y, x) \mid (x, y) \in P_2^{\mathcal{I}}\}$

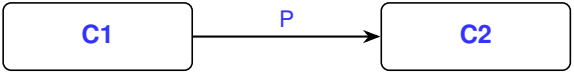
- We have used R to denote either an object property P or the inverse P^- of an object property.
- We have listed the axioms involving object properties, but OWL 2 QL allows for analogous axioms involving data properties.

RDF Schema – Domain of an object property

Domain of an object property

P **rdfs:domain** C1 .

$\exists P \sqsubseteq C_1$



When class C_1 is declared to be the **domain of object property** P, it means that, whenever a pair (o_1, o_2) is an instance of P, then o_1 must be an instance of C_1.
Said differently, the projection of P on its first component is a subclass of C_1.

Example: `:actsIn rdfs:domain :MovieActor .`

Inference: `<person/2> :actsIn <movie/3> .`

$\Longrightarrow$ `<person/2> rdf:type :MovieActor .`

In DL notation: $\exists \text{actsIn} \sqsubseteq \text{MovieActor}$

$\text{actsIn}(\textit{person/2}, \textit{movie/3}) \Longrightarrow \text{MovieActor}(\textit{person/2})$

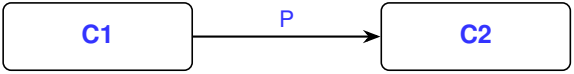
Note: In OWL 2 QL, the default cardinality for an object property is 0..*, as in our graphical notation. Hence, the above diagram without cardinalities captures correctly the situation where in OWL 2 QL we do not specify any cardinality for the object property.

RDF Schema – Domain of an object property

Domain of an object property

```
P rdfs:domain C1 .
```

$$\exists P \sqsubseteq C_1$$



When class C_1 is declared to be the **domain of object property** P, it means that, whenever a pair (o_1, o_2) is an instance of P, then o_1 must be an instance of C_1.
Said differently, the projection of P on its first component is a subclass of C_1.

Example: `:actsIn rdfs:domain :MovieActor .`
Inference: `<person/2> :actsIn <movie/3> .`

$$\Longrightarrow$$ `<person/2> rdf:type :MovieActor .`

In DL notation: $\exists \text{actsIn} \sqsubseteq \text{MovieActor}$

$\text{actsIn}(\textit{person/2}, \textit{movie/3}) \Longrightarrow \text{MovieActor}(\textit{person/2})$

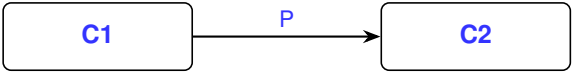
Note: In OWL 2 QL, the default cardinality for an object property is 0..*, as in our graphical notation. Hence, the above diagram without cardinalities captures correctly the situation where in OWL 2 QL we do not specify any cardinality for the object property.

RDF Schema – Domain of an object property

Domain of an object property

P **rdfs:domain** C1 .

$\exists P \sqsubseteq C_1$



When class C_1 is declared to be the **domain of object property** P, it means that, whenever a pair (o_1, o_2) is an instance of P, then o_1 must be an instance of C_1.
Said differently, the projection of P on its first component is a subclass of C_1.

Example: `:actsIn rdfs:domain :MovieActor .`
Inference: `<person/2> :actsIn <movie/3> .`
$\Longrightarrow$ `<person/2> rdf:type :MovieActor .`

In DL notation: $\exists \text{actsIn} \sqsubseteq \text{MovieActor}$
$\text{actsIn}(\textit{person/2}, \textit{movie/3}) \Longrightarrow \text{MovieActor}(\textit{person/2})$

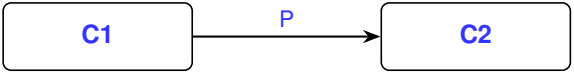
Note: In OWL 2 QL, the default cardinality for an object property is 0..*, as in our graphical notation. Hence, the above diagram without cardinalities captures correctly the situation where in OWL 2 QL we do not specify any cardinality for the object property.

RDF Schema – Domain of an object property

Domain of an object property

P **rdfs:domain** C1 .

$\exists P \sqsubseteq C_1$



When class C_1 is declared to be the **domain of object property** P, it means that, whenever a pair (o_1, o_2) is an instance of P, then o_1 must be an instance of C_1.
Said differently, the projection of P on its first component is a subclass of C_1.

Example: `:actsIn rdfs:domain :MovieActor .`
Inference: `<person/2> :actsIn <movie/3> .`
$\Longrightarrow$ `<person/2> rdf:type :MovieActor .`

In DL notation: $\exists$actsIn $\sqsubseteq$ MovieActor
actsIn(*person/2*, *movie/3*) $\Longrightarrow$ MovieActor(*person/2*)

Note: In OWL 2 QL, the default cardinality for an object property is 0..*, as in our graphical notation. Hence, the above diagram without cardinalities captures correctly the situation where in OWL 2 QL we do not specify any cardinality for the object property.

Syntax and semantics of OWL 2 QL KBs

Axiom type	OWL Syntax	DL Syntax	Semantics
Membership (class)	`<a> rdf:type <C>`	$C(a)$	$a \in C^{\mathcal{I}}$
Membership (data property)	`<a> :A <l>`	$A(a, \ell)$	$(a, \ell) \in A^{\mathcal{I}}$
Membership (object property)	`<a1> :P <a2>`	$P(a_1, a_2)$	$(a_1, a_2) \in P^{\mathcal{I}}$
Subclass assertion	`C1 rdfs:subClassOf C2`	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Class disjointness	`C1 owl:disjointWith C2`	$C_1 \sqsubseteq \neg C_2$	$C_1^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} - C_2^{\mathcal{I}}$
Property disjointness	`P1 owl:propertyDisjointWith P2`	$P_1 \sqsubseteq \neg P_2$	$P_1^{\mathcal{I}} \subseteq (\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) - P_2^{\mathcal{I}}$
Domain of a property	`P rdfs:domain C1`	$\exists P \sqsubseteq C_1$	$\{x \mid \exists y.(x, y) \in P^{\mathcal{I}}\} \subseteq C_1^{\mathcal{I}}$
Range of a property	`P rdfs:range C2`	$\exists P^- \sqsubseteq C_2$	$\{y \mid \exists x.(x, y) \in P^{\mathcal{I}}\} \subseteq C_2^{\mathcal{I}}$
Mandatory participation	using `owl:someValuesFrom`	$C \sqsubseteq \exists R$	$C^{\mathcal{I}} \subseteq \exists R^{\mathcal{I}}$
Subproperty assertion	`P1 rdfs:subPropertyOf R2`	$P_1 \sqsubseteq R_2$	$P_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
Inverse property	`P2 owl:inverseOf P1`	$P_1 \equiv P_2^-$	$P_1^{\mathcal{I}} = \{(y, x) \mid (x, y) \in P_2^{\mathcal{I}}\}$

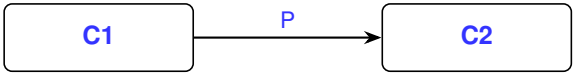
- We have used R to denote either an object property P or the inverse P^- of an object property.
- We have listed the axioms involving object properties, but OWL 2 QL allows for analogous axioms involving data properties.

RDF Schema – Range of an object property

Range of an object property

```
P rdfs:range C2 .
```

$$\exists P^- \sqsubseteq C_2$$



When class C_2 is declared to be the **range of object property** P, it means that, whenever a pair (o_1, o_2) is an instance of P, then o_2 must be an instance of C_2.
Said differently, the projection of P on its second component is a subclass of C_2.

Example: `:actsIn rdfs:range :Movie .`
Inference: `<person/2> :actsIn <movie/3> .`
$\Longrightarrow$ `<movie/3> rdf:type :Movie .`

In DL notation: $\exists \text{actsIn}^- \sqsubseteq \text{Movie}$
$\text{actsIn}(\textit{person/2}, \textit{movie/3}) \Longrightarrow \text{Movie}(\textit{movie/3})$

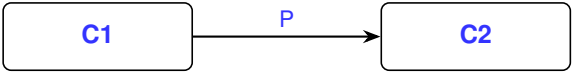
Note: In OWL 2 QL, the default cardinality for the inverse of an object property is 0..*, and this is also the default in our graphical notation. Hence, the above diagram captures correctly the situation where in OWL 2 QL we do not specify any cardinality for the inverse of an object property.

RDF Schema – Range of an object property

Range of an object property

```
P rdfs:range C2 .
```

$\exists P^- \sqsubseteq C_2$



When class C_2 is declared to be the **range of object property** P, it means that, whenever a pair (o_1, o_2) is an instance of P, then o_2 must be an instance of C_2.
Said differently, the projection of P on its second component is a subclass of C_2.

Example: `:actsIn rdfs:range :Movie .`
Inference: `<person/2> :actsIn <movie/3> .`
$\Longrightarrow$ `<movie/3> rdf:type :Movie .`

In DL notation: $\exists \text{actsIn}^- \sqsubseteq \text{Movie}$
$\text{actsIn}(\textit{person/2}, \textit{movie/3}) \Longrightarrow \text{Movie}(\textit{movie/3})$

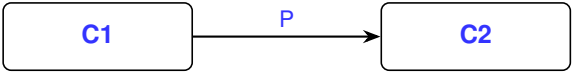
Note: In OWL 2 QL, the default cardinality for the inverse of an object property is 0..*, and this is also the default in our graphical notation. Hence, the above diagram captures correctly the situation where in OWL 2 QL we do not specify any cardinality for the inverse of an object property.

RDF Schema – Range of an object property

Range of an object property

```
P rdfs:range C2 .
```

$\exists P^- \sqsubseteq C_2$



When class C_2 is declared to be the **range of object property** P, it means that, whenever a pair (o_1, o_2) is an instance of P, then o_2 must be an instance of C_2.
Said differently, the projection of P on its second component is a subclass of C_2.

Example: `:actsIn rdfs:range :Movie .`
Inference: `<person/2> :actsIn <movie/3> .`
$\Longrightarrow$ `<movie/3> rdf:type :Movie .`

In DL notation: $\exists \text{actsIn}^- \sqsubseteq \text{Movie}$
$\text{actsIn}(\textit{person/2}, \textit{movie/3}) \Longrightarrow \text{Movie}(\textit{movie/3})$

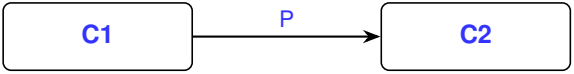
Note: In OWL 2 QL, the default cardinality for the inverse of an object property is 0..*, and this is also the default in our graphical notation. Hence, the above diagram captures correctly the situation where in OWL 2 QL we do not specify any cardinality for the inverse of an object property.

RDF Schema – Range of an object property

Range of an object property

```
P rdfs:range C2 .
```

$\exists P^- \sqsubseteq C_2$



When class C_2 is declared to be the **range of object property** P, it means that, whenever a pair (o_1, o_2) is an instance of P, then o_2 must be an instance of C_2.
Said differently, the projection of P on its second component is a subclass of C_2.

Example: `:actsIn rdfs:range :Movie .`
Inference: `<person/2> :actsIn <movie/3> .`
$\Longrightarrow$ `<movie/3> rdf:type :Movie .`

In DL notation: $\exists \text{actsIn}^- \sqsubseteq \text{Movie}$
$\text{actsIn}(\textit{person/2}, \textit{movie/3}) \Longrightarrow \text{Movie}(\textit{movie/3})$

Note: In OWL 2 QL, the default cardinality for the inverse of an object property is 0..*, and this is also the default in our graphical notation. Hence, the above diagram captures correctly the situation where in OWL 2 QL we do not specify any cardinality for the inverse of an object property.

Syntax and semantics of OWL 2 QL KBs

Axiom type	OWL Syntax	DL Syntax	Semantics
Membership (class)	`<a> rdf:type <C>`	$C(a)$	$a \in C^{\mathcal{I}}$
Membership (data property)	`<a> :A <l>`	$A(a, \ell)$	$(a, \ell) \in A^{\mathcal{I}}$
Membership (object property)	`<a1> :P <a2>`	$P(a_1, a_2)$	$(a_1, a_2) \in P^{\mathcal{I}}$
Subclass assertion	`C1 rdfs:subClassOf C2`	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Class disjointness	`C1 owl:disjointWith C2`	$C_1 \sqsubseteq \neg C_2$	$C_1^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} - C_2^{\mathcal{I}}$
Property disjointness	`P1 owl:propertyDisjointWith P2`	$P_1 \sqsubseteq \neg P_2$	$P_1^{\mathcal{I}} \subseteq (\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) - P_2^{\mathcal{I}}$
Domain of a property	`P rdfs:domain C1`	$\exists P \sqsubseteq C_1$	$\{x \mid \exists y.(x, y) \in P^{\mathcal{I}}\} \subseteq C_1^{\mathcal{I}}$
Range of a property	`P rdfs:range C2`	$\exists P^{-} \sqsubseteq C_2$	$\{y \mid \exists x.(x, y) \in P^{\mathcal{I}}\} \subseteq C_2^{\mathcal{I}}$
Mandatory participation	using `owl:someValuesFrom`	$C \sqsubseteq \exists R$	$C^{\mathcal{I}} \subseteq \exists R^{\mathcal{I}}$
Subproperty assertion	`P1 rdfs:subPropertyOf R2`	$P_1 \sqsubseteq R_2$	$P_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
Inverse property	`P2 owl:inverseOf P1`	$P_1 \equiv P_2^{-}$	$P_1^{\mathcal{I}} = \{(y, x) \mid (x, y) \in P_2^{\mathcal{I}}\}$

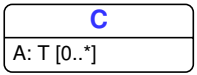
- We have used R to denote either an object property P or the inverse P^{-} of an object property.
- We have listed the axioms involving object properties, but OWL 2 QL allows for analogous axioms involving data properties.

RDF Schema – Domain and range of a data property

Domain and range of a data property

`A rdfs:domain C .` $\quad\quad$ `A rdfs:range T .`

$\exists A \sqsubseteq C$ $\quad\quad$ $\exists A^- \sqsubseteq T$



The declaration of the domain of a data property has the same meaning as for object properties. As for the range, notice that the RDFS statement "`A rdfs:range T.`" is analogous to the one for object properties.

Example: `:title rdfs:domain :Movie .`
`:title rdfs:range xsd:string .`

Inference: `<movie/3> :title "Bladerunner" .`
$\Longrightarrow$ `<movie/3> rdf:type :Movie .`
`"Bladerunner"` is of type `xsd:string`

In DL notation: $\exists \text{title} \sqsubseteq \text{Movie}$ $\quad\quad$ $\text{Movie} \sqsubseteq \forall \text{title}.\text{String}$

$\text{title}(\textit{movie/3}, \texttt{"Bladerunner"}) \Longrightarrow \text{Movie}(\textit{movie/3})$

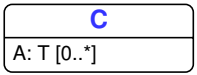
Note: In OWL 2 QL, the default cardinality for a data property is [0..*], while in our graphical notation we assume [1..1] as the default. Hence, the above diagram captures correctly the situation where in OWL 2 QL we do not specify any cardinality for the data property.

RDF Schema – Domain and range of a data property

Domain and range of a data property

`A rdfs:domain C .` $\quad\quad$ `A rdfs:range T .`

$\exists A \sqsubseteq C$ $\quad\quad$ $\exists A^- \sqsubseteq T$



The declaration of the domain of a data property has the same meaning as for object properties. As for the range, notice that the RDFS statement "`A rdfs:range T.`" is analogous to the one for object properties.

Example: `:title rdfs:domain :Movie .`
`:title rdfs:range xsd:string .`

Inference: `<movie/3> :title "Bladerunner" .`
$\Longrightarrow$ `<movie/3> rdf:type :Movie .`
`"Bladerunner"` is of type `xsd:string`

In DL notation: $\exists \text{title} \sqsubseteq \text{Movie}$ $\quad\quad$ $\text{Movie} \sqsubseteq \forall \text{title}.\text{String}$

$\text{title}(\textit{movie/3}, \texttt{"Bladerunner"}) \Longrightarrow \text{Movie}(\textit{movie/3})$

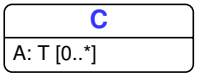
Note: In OWL 2 QL, the default cardinality for a data property is [0..*], while in our graphical notation we assume [1..1] as the default. Hence, the above diagram captures correctly the situation where in OWL 2 QL we do not specify any cardinality for the data property.

RDF Schema – Domain and range of a data property

Domain and range of a data property

`A rdfs:domain C .` $\quad\quad$ `A rdfs:range T .`

$\exists A \sqsubseteq C$ $\quad\quad$ $\exists A^- \sqsubseteq T$



The declaration of the domain of a data property has the same meaning as for object properties. As for the range, notice that the RDFS statement "`A rdfs:range T.`" is analogous to the one for object properties.

Example: `:title rdfs:domain :Movie .`
`:title rdfs:range xsd:string .`

Inference: `<movie/3> :title "Bladerunner" .`
$\Longrightarrow$ `<movie/3> rdf:type :Movie .`
`"Bladerunner"` is of type `xsd:string`

In DL notation: $\exists$title $\sqsubseteq$ Movie $\quad$ Movie $\sqsubseteq \forall$title.String
title(*movie/3*, `"Bladerunner"`) $\Longrightarrow$ Movie(*movie/3*)

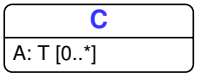
Note: In OWL 2 QL, the default cardinality for a data property is [0..*], while in our graphical notation we assume [1..1] as the default. Hence, the above diagram captures correctly the situation where in OWL 2 QL we do not specify any cardinality for the data property.

RDF Schema – Domain and range of a data property

Domain and range of a data property

`A rdfs:domain C .` $\quad\quad$ `A rdfs:range T .`

$\exists A \sqsubseteq C$ $\quad\quad$ $\exists A^- \sqsubseteq T$



The declaration of the domain of a data property has the same meaning as for object properties. As for the range, notice that the RDFS statement "`A rdfs:range T.`" is analogous to the one for object properties.

Example: `:title rdfs:domain :Movie .`
`:title rdfs:range xsd:string .`

Inference: `<movie/3> :title "Bladerunner" .`
$\Longrightarrow$ `<movie/3> rdf:type :Movie .`
`"Bladerunner"` is of type `xsd:string`

In DL notation: $\exists$title $\sqsubseteq$ Movie $\quad$ Movie $\sqsubseteq \forall$title.String
title(*movie/3*, `"Bladerunner"`) $\Longrightarrow$ Movie(*movie/3*)

Note: In OWL 2 QL, the default cardinality for a data property is [0..*], while in our graphical notation we assume [1..1] as the default. Hence, the above diagram captures correctly the situation where in OWL 2 QL we do not specify any cardinality for the data property.

Domain and range of properties – OWL 2 QL vs. graphical notation

- Note that in our graphical notation, whenever we specify an object property, we need to connect two classes, and therefore we are implicitly specifying the domain and range of the property.
- Something analogous holds for data properties, since we specify them within a class, and hence we fix their domain.
- In OWL 2 QL, instead, we are not forced to specify the domain or the range of properties. We can simply declare them, and leave them completely unconstrained.
- When a data property is unconstrained, this means that:
  - the domain is `owl:Thing`, which is the class of all objects;
  - the range is `rdfs:Literal`, which denotes the set of all possible literals.
- When an object property is unconstrained, this means that both its domain and its range are `owl:Thing`.

Syntax and semantics of OWL 2 QL KBs

Axiom type	OWL Syntax	DL Syntax	Semantics
Membership (class)	`<a> rdf:type <C>`	$C(a)$	$a \in C^{\mathcal{I}}$
Membership (data property)	`<a> :A <l>`	$A(a, \ell)$	$(a, \ell) \in A^{\mathcal{I}}$
Membership (object property)	`<a1> :P <a2>`	$P(a_1, a_2)$	$(a_1, a_2) \in P^{\mathcal{I}}$
Subclass assertion	`C1 rdfs:subClassOf C2`	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Class disjointness	`C1 owl:disjointWith C2`	$C_1 \sqsubseteq \neg C_2$	$C_1^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} - C_2^{\mathcal{I}}$
Property disjointness	`P1 owl:propertyDisjointWith P2`	$P_1 \sqsubseteq \neg P_2$	$P_1^{\mathcal{I}} \subseteq (\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) - P_2^{\mathcal{I}}$
Domain of a property	`P rdfs:domain C1`	$\exists P \sqsubseteq C_1$	$\{x \mid \exists y.(x, y) \in P^{\mathcal{I}}\} \subseteq C_1^{\mathcal{I}}$
Range of a property	`P rdfs:range C2`	$\exists P^{-} \sqsubseteq C_2$	$\{y \mid \exists x.(x, y) \in P^{\mathcal{I}}\} \subseteq C_2^{\mathcal{I}}$
Mandatory participation	using `owl:someValuesFrom`	$C \sqsubseteq \exists R$	$C^{\mathcal{I}} \subseteq \exists R^{\mathcal{I}}$
Subproperty assertion	`P1 rdfs:subPropertyOf R2`	$P_1 \sqsubseteq R_2$	$P_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
Inverse property	`P2 owl:inverseOf P1`	$P_1 \equiv P_2^{-}$	$P_1^{\mathcal{I}} = \{(y, x) \mid (x, y) \in P_2^{\mathcal{I}}\}$

- We have used R to denote either an object property P or the inverse P^{-} of an object property.
- We have listed the axioms involving object properties, but OWL 2 QL allows for analogous axioms involving data properties.

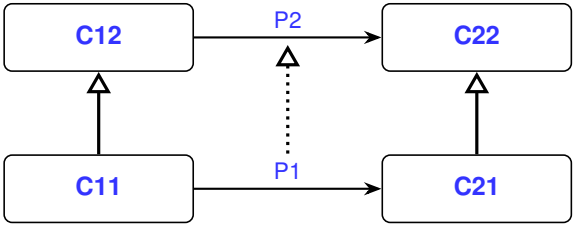
RDF Schema – Property hierarchy

Property hierarchy

```
P1 rdfs:subPropertyOf P2 .
```

$$P_1 \sqsubseteq P_2$$

When a property P_1 is declared to be a **sub-property of** P_2, then every pair of objects that is an instance of P_1 is also an instance of P_2.



Note: Typically, when a property P_1 is a sub-property of a property P_2, then the respective domains and ranges are in a subclass relationship.

Example: `:actsIn rdfs:subPropertyOf :playsIn .`

Inference: `<person/2> :actsIn <movie/3> .`
$\Longrightarrow$ `<person/2> :playsIn <movie/3> .`

In DL notation: $\mathit{actsIn} \sqsubseteq \mathit{playsIn}$

$\mathit{actsIn}(\mathit{person/2}, \mathit{movie/3}) \Longrightarrow \mathit{playsIn}(\mathit{person/2}, \mathit{movie/3})$

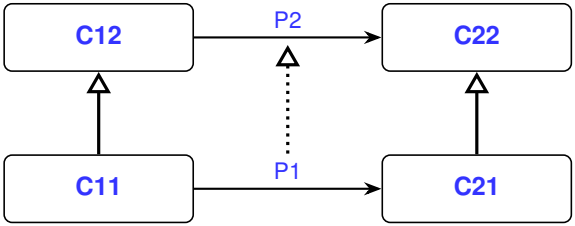
RDF Schema – Property hierarchy

Property hierarchy

```
P1 rdfs:subPropertyOf P2 .
```

$$P_1 \sqsubseteq P_2$$

When a property P_1 is declared to be a **sub-property of** P_2, then every pair of objects that is an instance of P_1 is also an instance of P_2.



Note: Typically, when a property P_1 is a sub-property of a property P_2, then the respective domains and ranges are in a subclass relationship.

Example: `:actsIn rdfs:subPropertyOf :playsIn .`

Inference: `<person/2> :actsIn <movie/3> .`
$\Longrightarrow$ `<person/2> :playsIn <movie/3> .`

In DL notation: $actsIn \sqsubseteq playsIn$

$actsIn(person/2, movie/3) \Longrightarrow playsIn(person/2, movie/3)$

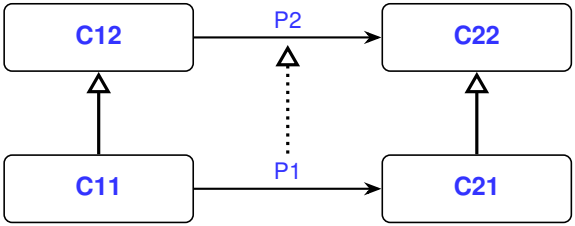
RDF Schema – Property hierarchy

Property hierarchy

```
P1 rdfs:subPropertyOf P2 .
```

$$P_1 \sqsubseteq P_2$$

When a property P_1 is declared to be a **sub-property of** P_2, then every pair of objects that is an instance of P_1 is also an instance of P_2.



Note: Typically, when a property P_1 is a sub-property of a property P_2, then the respective domains and ranges are in a subclass relationship.

Example: `:actsIn rdfs:subPropertyOf :playsIn .`

Inference: `<person/2> :actsIn <movie/3> .`

$\Longrightarrow$ `<person/2> :playsIn <movie/3> .`

In DL notation: $\mathit{actsIn} \sqsubseteq \mathit{playsIn}$

$\mathit{actsIn}(\mathit{person/2}, \mathit{movie/3}) \Longrightarrow \mathit{playsIn}(\mathit{person/2}, \mathit{movie/3})$

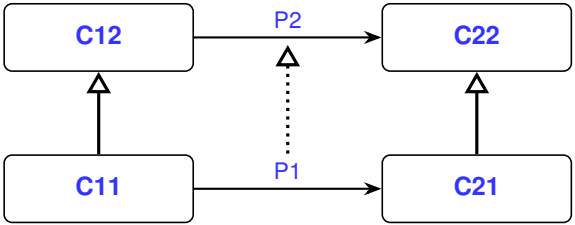
RDF Schema – Property hierarchy

Property hierarchy

```
P1 rdfs:subPropertyOf P2 .
```

$$P_1 \sqsubseteq P_2$$

When a property P_1 is declared to be a **sub-property of** P_2, then every pair of objects that is an instance of P_1 is also an instance of P_2.



Note: Typically, when a property P_1 is a sub-property of a property P_2, then the respective domains and ranges are in a subclass relationship.

Example: `:actsIn rdfs:subPropertyOf :playsIn .`

Inference: `<person/2> :actsIn <movie/3> .`

$\Longrightarrow$ `<person/2> :playsIn <movie/3> .`

In DL notation: $\text{actsIn} \sqsubseteq \text{playsIn}$

$\text{actsIn}(\textit{person/2}, \textit{movie/3}) \Longrightarrow \text{playsIn}(\textit{person/2}, \textit{movie/3})$

Syntax and semantics of OWL 2 QL KBs

Axiom type	OWL Syntax	DL Syntax	Semantics
Membership (class)	`<a> rdf:type <C>`	$C(a)$	$a \in C^{\mathcal{I}}$
Membership (data property)	`<a> :A <l>`	$A(a, \ell)$	$(a, \ell) \in A^{\mathcal{I}}$
Membership (object property)	`<a1> :P <a2>`	$P(a_1, a_2)$	$(a_1, a_2) \in P^{\mathcal{I}}$
Subclass assertion	`C1 rdfs:subClassOf C2`	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Class disjointness	`C1 owl:disjointWith C2`	$C_1 \sqsubseteq \neg C_2$	$C_1^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} - C_2^{\mathcal{I}}$
Property disjointness	`P1 owl:propertyDisjointWith P2`	$P_1 \sqsubseteq \neg P_2$	$P_1^{\mathcal{I}} \subseteq (\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) - P_2^{\mathcal{I}}$
Domain of a property	`P rdfs:domain C1`	$\exists P \sqsubseteq C_1$	$\{x \mid \exists y.(x, y) \in P^{\mathcal{I}}\} \subseteq C_1^{\mathcal{I}}$
Range of a property	`P rdfs:range C2`	$\exists P^{-} \sqsubseteq C_2$	$\{y \mid \exists x.(x, y) \in P^{\mathcal{I}}\} \subseteq C_2^{\mathcal{I}}$
Mandatory participation	using `owl:someValuesFrom`	$C \sqsubseteq \exists R$	$C^{\mathcal{I}} \subseteq \exists R^{\mathcal{I}}$
Subproperty assertion	`P1 rdfs:subPropertyOf R2`	$P_1 \sqsubseteq R_2$	$P_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
Inverse property	`P2 owl:inverseOf P1`	$P_1 \equiv P_2^{-}$	$P_1^{\mathcal{I}} = \{(y, x) \mid (x, y) \in P_2^{\mathcal{I}}\}$

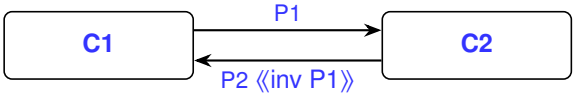
- We have used R to denote either an object property P or the inverse P^{-} of an object property.
- We have listed the axioms involving object properties, but OWL 2 QL allows for analogous axioms involving data properties.

OWL 2 QL – Inverse (object) property

Inverse property

P2 **owl:inverseOf** P1 .

$P_2^- \sqsubseteq P_1$ and $P_1^- \sqsubseteq P_2$



When a property P_2 is declared to be the **inverse of** P_1, we have that, (o_1, o_2) is an instance of P_2 if and only if (o_2, o_1) is an instance of P_1.

Note: In the graphical notation that we adopt, there is no standard way to represent that one object property is the inverse of another one. Therefore, we have introduced a notation resembling the one used for stereotypes in UML.

Example: `:playsIn owl:inverseOf :hasActor .`

Inference: `<person/2> :playsIn <movie/3> .`

$\Longrightarrow$ `<movie/3> :hasActor <person/2> .`

In DL notation: playsIn $\sqsubseteq$ hasActor$^-$ and hasActor$^-$ $\sqsubseteq$ playsIn

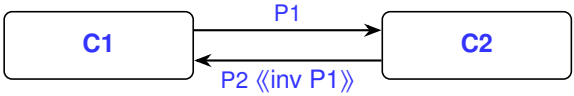
playsIn(*person/2*, *movie/3*) $\Longrightarrow$ hasActor(*movie/3*, *person/2*)

OWL 2 QL – Inverse (object) property

Inverse property

P2 **owl:inverseOf** P1 .

$P_2^- \sqsubseteq P_1$ and $P_1^- \sqsubseteq P_2$



When a property P_2 is declared to be the **inverse of** P_1, we have that, (o_1, o_2) is an instance of P_2 if and only if (o_2, o_1) is an instance of P_1.

Note: In the graphical notation that we adopt, there is no standard way to represent that one object property is the inverse of another one. Therefore, we have introduced a notation resembling the one used for stereotypes in UML.

Example: :playsIn **owl:inverseOf** :hasActor .

Inference: <person/2> :playsIn <movie/3> .

$\Longrightarrow$ <movie/3> :hasActor <person/2> .

In DL notation: playsIn $\sqsubseteq$ hasActor$^-$ and hasActor$^-$ $\sqsubseteq$ playsIn

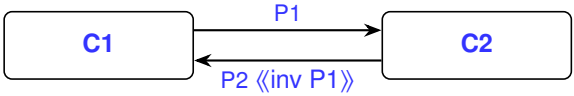
playsIn(*person/2, movie/3*) $\Longrightarrow$ hasActor(*movie/3, person/2*)

OWL 2 QL – Inverse (object) property

Inverse property

P2 owl:inverseOf P1 .

$P_2^- \sqsubseteq P_1$ and $P_1^- \sqsubseteq P_2$



When a property P_2 is declared to be the **inverse of** P_1, we have that, (o_1, o_2) is an instance of P_2 if and only if (o_2, o_1) is an instance of P_1.

Note: In the graphical notation that we adopt, there is no standard way to represent that one object property is the inverse of another one. Therefore, we have introduced a notation resembling the one used for stereotypes in UML.

Example: :playsIn owl:inverseOf :hasActor .

Inference: <person/2> :playsIn <movie/3> .

$\Longrightarrow$ <movie/3> :hasActor <person/2> .

In DL notation: playsIn $\sqsubseteq$ hasActor$^-$ and hasActor$^-$ $\sqsubseteq$ playsIn

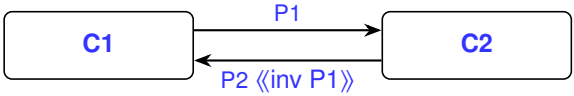
playsIn(*person/2*, *movie/3*) $\Longrightarrow$ hasActor(*movie/3*, *person/2*)

OWL 2 QL – Inverse (object) property

Inverse property

P2 **owl:inverseOf** P1 .

$P_2^- \sqsubseteq P_1$ and $P_1^- \sqsubseteq P_2$



When a property P_2 is declared to be the **inverse of** P_1, we have that, (o_1, o_2) is an instance of P_2 if and only if (o_2, o_1) is an instance of P_1.

Note: In the graphical notation that we adopt, there is no standard way to represent that one object property is the inverse of another one. Therefore, we have introduced a notation resembling the one used for stereotypes in UML.

Example: `:playsIn owl:inverseOf :hasActor .`

Inference: `<person/2> :playsIn <movie/3> .`

$\Longrightarrow$ `<movie/3> :hasActor <person/2> .`

In DL notation: playsIn $\sqsubseteq$ hasActor$^-$ and hasActor$^-$ $\sqsubseteq$ playsIn

playsIn(*person/2*, *movie/3*) $\Longrightarrow$ hasActor(*movie/3*, *person/2*)

Syntax and semantics of OWL 2 QL KBs

Axiom type	OWL Syntax	DL Syntax	Semantics
Membership (class)	`<a> rdf:type <C>`	$C(a)$	$a \in C^{\mathcal{I}}$
Membership (data property)	`<a> :A <l>`	$A(a, \ell)$	$(a, \ell) \in A^{\mathcal{I}}$
Membership (object property)	`<a1> :P <a2>`	$P(a_1, a_2)$	$(a_1, a_2) \in P^{\mathcal{I}}$
Subclass assertion	`C1 rdfs:subClassOf C2`	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Class disjointness	`C1 owl:disjointWith C2`	$C_1 \sqsubseteq \neg C_2$	$C_1^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} - C_2^{\mathcal{I}}$
Property disjointness	`P1 owl:propertyDisjointWith P2`	$P_1 \sqsubseteq \neg P_2$	$P_1^{\mathcal{I}} \subseteq (\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) - P_2^{\mathcal{I}}$
Domain of a property	`P rdfs:domain C1`	$\exists P \sqsubseteq C_1$	$\{x \mid \exists y.(x, y) \in P^{\mathcal{I}}\} \subseteq C_1^{\mathcal{I}}$
Range of a property	`P rdfs:range C2`	$\exists P^- \sqsubseteq C_2$	$\{y \mid \exists x.(x, y) \in P^{\mathcal{I}}\} \subseteq C_2^{\mathcal{I}}$
Mandatory participation	using `owl:someValuesFrom`	$C \sqsubseteq \exists R$	$C^{\mathcal{I}} \subseteq \exists R^{\mathcal{I}}$
Subproperty assertion	`P1 rdfs:subPropertyOf R2`	$P_1 \sqsubseteq R_2$	$P_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
Inverse property	`P2 owl:inverseOf P1`	$P_1 \equiv P_2^-$	$P_1^{\mathcal{I}} = \{(y, x) \mid (x, y) \in P_2^{\mathcal{I}}\}$

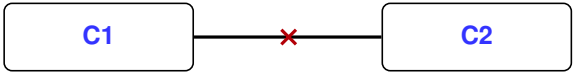
- We have used R to denote either an object property P or the inverse P^- of an object property.
- We have listed the axioms involving object properties, but OWL 2 QL allows for analogous axioms involving data properties.

OWL 2 QL – Class disjointness

Class disjointness

```
C1 owl:disjointWith C2 .
```

$C_1 \sqsubseteq \neg C_2$



When two classes C_1 and C_2 are declared to be **disjoint**, then they can have no instances in common. I.e., if o is an instance of C_1, then it is not an instance of C_2, and vice-versa.

Note: In the graphical notation that we adopt, there is no standard way to represent that two classes are disjoint. Therefore, we have introduced a convenient graphical construct.
Moreover, when representing an ontology as a diagram, **we assume that two classes that do not belong to the same ISA hierarchy are disjoint**.

Example: `:Actor owl:disjointWith :Movie .`
Inference: `<person/2> rdf:type :Actor .`
`<person/2> rdf:type :Movie .`
$\Longrightarrow$ RDF graph inconsistent with the ontology

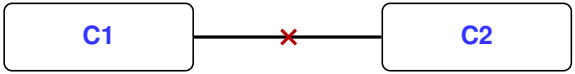
In DL notation: $\text{Actor} \sqsubseteq \neg\text{Movie}$
$\text{Actor}(\textit{person/2}),\ \text{Movie}(\textit{person/2}) \Longrightarrow$ RDF graph inconsistent with the ontology

OWL 2 QL – Class disjointness

Class disjointness

```
C1 owl:disjointWith C2 .
```

$C_1 \sqsubseteq \neg C_2$



When two classes C_1 and C_2 are declared to be **disjoint**, then they can have no instances in common. I.e., if o is an instance of C_1, then it is not an instance of C_2, and vice-versa.

Note: In the graphical notation that we adopt, there is no standard way to represent that two classes are disjoint. Therefore, we have introduced a convenient graphical construct.
Moreover, when representing an ontology as a diagram, **we assume that two classes that do not belong to the same ISA hierarchy are disjoint**.

Example: `:Actor owl:disjointWith :Movie .`
Inference: `<person/2> rdf:type :Actor .`
`<person/2> rdf:type :Movie .`
$\Longrightarrow$ RDF graph inconsistent with the ontology

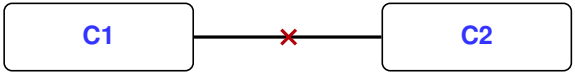
In DL notation: $\text{Actor} \sqsubseteq \neg\text{Movie}$
$\text{Actor}(\textit{person/2}),\ \text{Movie}(\textit{person/2}) \Longrightarrow$ RDF graph inconsistent with the ontology

OWL 2 QL – Class disjointness

Class disjointness

```
C1 owl:disjointWith C2 .
```

$C_1 \sqsubseteq \neg C_2$



When two classes C_1 and C_2 are declared to be **disjoint**, then they can have no instances in common. I.e., if o is an instance of C_1, then it is not an instance of C_2, and vice-versa.

Note: In the graphical notation that we adopt, there is no standard way to represent that two classes are disjoint. Therefore, we have introduced a convenient graphical construct.
Moreover, when representing an ontology as a diagram, **we assume that two classes that do not belong to the same ISA hierarchy are disjoint**.

Example: `:Actor owl:disjointWith :Movie .`
Inference: `<person/2> rdf:type :Actor .`
`<person/2> rdf:type :Movie .`
$\Longrightarrow$ RDF graph inconsistent with the ontology

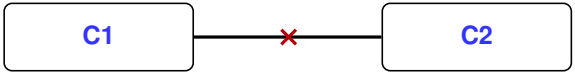
In DL notation: $\text{Actor} \sqsubseteq \neg\text{Movie}$
$\text{Actor}(person/2)$, $\text{Movie}(person/2) \Longrightarrow$ RDF graph inconsistent with the ontology

OWL 2 QL – Class disjointness

Class disjointness

```
C1 owl:disjointWith C2 .
```

$C_1 \sqsubseteq \neg C_2$



When two classes C_1 and C_2 are declared to be **disjoint**, then they can have no instances in common. I.e., if o is an instance of C_1, then it is not an instance of C_2, and vice-versa.

Note: In the graphical notation that we adopt, there is no standard way to represent that two classes are disjoint. Therefore, we have introduced a convenient graphical construct.
Moreover, when representing an ontology as a diagram, **we assume that two classes that do not belong to the same ISA hierarchy are disjoint**.

Example: `:Actor owl:disjointWith :Movie .`
Inference:
```
<person/2> rdf:type :Actor .
<person/2> rdf:type :Movie .
```
$\Longrightarrow$ RDF graph inconsistent with the ontology

In DL notation: $\text{Actor} \sqsubseteq \neg\text{Movie}$
$\text{Actor}(person/2)$, $\text{Movie}(person/2)$ $\Longrightarrow$ RDF graph inconsistent with the ontology

Syntax and semantics of OWL 2 QL KBs

Axiom type	OWL Syntax	DL Syntax	Semantics
Membership (class)	`<a> rdf:type <C>`	$C(a)$	$a \in C^{\mathcal{I}}$
Membership (data property)	`<a> :A <l>`	$A(a, \ell)$	$(a, \ell) \in A^{\mathcal{I}}$
Membership (object property)	`<a1> :P <a2>`	$P(a_1, a_2)$	$(a_1, a_2) \in P^{\mathcal{I}}$
Subclass assertion	`C1 rdfs:subClassOf C2`	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Class disjointness	`C1 owl:disjointWith C2`	$C_1 \sqsubseteq \neg C_2$	$C_1^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} - C_2^{\mathcal{I}}$
Property disjointness	`P1 owl:propertyDisjointWith P2`	$P_1 \sqsubseteq \neg P_2$	$P_1^{\mathcal{I}} \subseteq (\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) - P_2^{\mathcal{I}}$
Domain of a property	`P rdfs:domain C1`	$\exists P \sqsubseteq C_1$	$\{x \mid \exists y.(x, y) \in P^{\mathcal{I}}\} \subseteq C_1^{\mathcal{I}}$
Range of a property	`P rdfs:range C2`	$\exists P^- \sqsubseteq C_2$	$\{y \mid \exists x.(x, y) \in P^{\mathcal{I}}\} \subseteq C_2^{\mathcal{I}}$
Mandatory participation	using `owl:someValuesFrom`	$C \sqsubseteq \exists R$	$C^{\mathcal{I}} \subseteq \exists R^{\mathcal{I}}$
Subproperty assertion	`P1 rdfs:subPropertyOf R2`	$P_1 \sqsubseteq R_2$	$P_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
Inverse property	`P2 owl:inverseOf P1`	$P_1 \equiv P_2^-$	$P_1^{\mathcal{I}} = \{(y, x) \mid (x, y) \in P_2^{\mathcal{I}}\}$

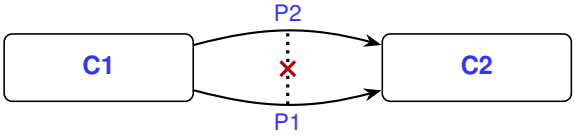
- We have used R to denote either an object property P or the inverse P^- of an object property.
- We have listed the axioms involving object properties, but OWL 2 QL allows for analogous axioms involving data properties, except that we might not use the inverse of a data property.

OWL 2 QL – Property disjointness

Property disjointness

```
P1 owl:propertyDisjointWith P2 .
```

$$P_1 \sqsubseteq \neg P_2$$



When two properties P_1 and P_2 are declared to be **disjoint**, they can have no instances in common.

Note: In the graphical notation that we adopt, there is no standard way to represent that two object properties are dijoint. Therefore, we have introduced a convenient graphical construct.
When the domain or the range of two properties are disjoint, then so are the properties. On the other hand, there might be two properties that are disjoint, although their domain and range are not.

Example: `:departFrom owl:propertyDisjointWith :arriveIn .`
Inference: `<flight/1> :departFrom <airport/5> .`
`<flight/1> :arriveIn <airport/5> .`
$\Longrightarrow$ RDF graph inconsistent with the ontology

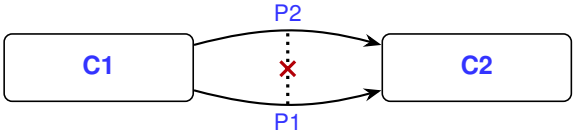
In DL notation: $\text{departFrom} \sqsubseteq \neg\text{arriveIn}$
$\text{departFrom}(\textit{flight/1}, \textit{airport/5})$, $\text{arriveIn}(\textit{flight/1}, \textit{airport/5})$
$\Longrightarrow$ RDF graph inconsistent with the ontology

OWL 2 QL – Property disjointness

Property disjointness

```
P1 owl:propertyDisjointWith P2 .
```

$$P_1 \sqsubseteq \neg P_2$$



When two properties P_1 and P_2 are declared to be **disjoint**, they can have no instances in common.

Note: In the graphical notation that we adopt, there is no standard way to represent that two object properties are dijoint. Therefore, we have introduced a convenient graphical construct.
When the domain or the range of two properties are disjoint, then so are the properties. On the other hand, there might be two properties that are disjoint, although their domain and range are not.

Example: `:departFrom owl:propertyDisjointWith :arriveIn .`

Inference:
```
<flight/1> :departFrom <airport/5> .
<flight/1> :arriveIn <airport/5> .
```
$\Longrightarrow$ RDF graph inconsistent with the ontology

In DL notation: $\mathit{departFrom} \sqsubseteq \neg \mathit{arriveIn}$

$\mathit{departFrom}(\mathit{flight/1}, \mathit{airport/5})$, $\mathit{arriveIn}(\mathit{flight/1}, \mathit{airport/5})$

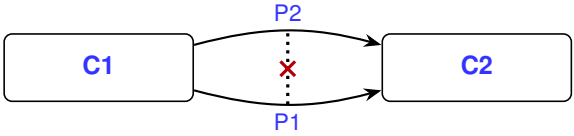
$\Longrightarrow$ RDF graph inconsistent with the ontology

OWL 2 QL – Property disjointness

Property disjointness

```
P1 owl:propertyDisjointWith P2 .
```

$$P_1 \sqsubseteq \neg P_2$$



When two properties P_1 and P_2 are declared to be **disjoint**, they can have no instances in common.

Note: In the graphical notation that we adopt, there is no standard way to represent that two object properties are dijoint. Therefore, we have introduced a convenient graphical construct.
When the domain or the range of two properties are disjoint, then so are the properties. On the other hand, there might be two properties that are disjoint, although their domain and range are not.

Example: `:departFrom owl:propertyDisjointWith :arriveIn .`
Inference:
```
<flight/1> :departFrom <airport/5> .
<flight/1> :arriveIn <airport/5> .
```
$\Longrightarrow$ RDF graph inconsistent with the ontology

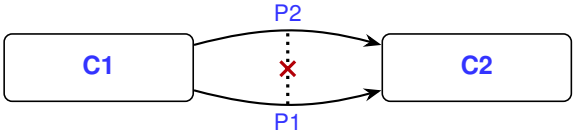
In DL notation: $\mathsf{departFrom} \sqsubseteq \neg \mathsf{arriveIn}$
$\mathsf{departFrom}(\mathit{flight/1}, \mathit{airport/5}), \ \mathsf{arriveIn}(\mathit{flight/1}, \mathit{airport/5})$
$\Longrightarrow$ RDF graph inconsistent with the ontology

OWL 2 QL – Property disjointness

Property disjointness

```
P1 owl:propertyDisjointWith P2 .
```

$$P_1 \sqsubseteq \neg P_2$$



When two properties P_1 and P_2 are declared to be **disjoint**, they can have no instances in common.

Note: In the graphical notation that we adopt, there is no standard way to represent that two object properties are dijoint. Therefore, we have introduced a convenient graphical construct.
When the domain or the range of two properties are disjoint, then so are the properties. On the other hand, there might be two properties that are disjoint, although their domain and range are not.

Example: `:departFrom owl:propertyDisjointWith :arriveIn .`
Inference:
```
<flight/1> :departFrom <airport/5> .
<flight/1> :arriveIn <airport/5> .
```
$\Longrightarrow$ RDF graph inconsistent with the ontology

In DL notation: $\mathsf{departFrom} \sqsubseteq \neg \mathsf{arriveIn}$
$\mathsf{departFrom}(\mathit{flight/1}, \mathit{airport/5})$, $\mathsf{arriveIn}(\mathit{flight/1}, \mathit{airport/5})$
$\Longrightarrow$ RDF graph inconsistent with the ontology

Syntax and semantics of OWL 2 QL KBs

Axiom type	OWL Syntax	DL Syntax	Semantics
Membership (class)	`<a> rdf:type <C>`	$C(a)$	$a \in C^{\mathcal{I}}$
Membership (data property)	`<a> :A <l>`	$A(a, \ell)$	$(a, \ell) \in A^{\mathcal{I}}$
Membership (object property)	`<a1> :P <a2>`	$P(a_1, a_2)$	$(a_1, a_2) \in P^{\mathcal{I}}$
Subclass assertion	`C1 rdfs:subClassOf C2`	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Class disjointness	`C1 owl:disjointWith C2`	$C_1 \sqsubseteq \neg C_2$	$C_1^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} - C_2^{\mathcal{I}}$
Property disjointness	`P1 owl:propertyDisjointWith P2`	$P_1 \sqsubseteq \neg P_2$	$P_1^{\mathcal{I}} \subseteq (\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) - P_2^{\mathcal{I}}$
Domain of a property	`P rdfs:domain C1`	$\exists P \sqsubseteq C_1$	$\{x \mid \exists y.(x, y) \in P^{\mathcal{I}}\} \subseteq C_1^{\mathcal{I}}$
Range of a property	`P rdfs:range C2`	$\exists P^- \sqsubseteq C_2$	$\{y \mid \exists x.(x, y) \in P^{\mathcal{I}}\} \subseteq C_2^{\mathcal{I}}$
Mandatory participation	using `owl:someValuesFrom`	$C \sqsubseteq \exists R$	$C^{\mathcal{I}} \subseteq \exists R^{\mathcal{I}}$
Subproperty assertion	`P1 rdfs:subPropertyOf R2`	$P_1 \sqsubseteq R_2$	$P_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
Inverse property	`P2 owl:inverseOf P1`	$P_1 \equiv P_2^-$	$P_1^{\mathcal{I}} = \{(y, x) \mid (x, y) \in P_2^{\mathcal{I}}\}$

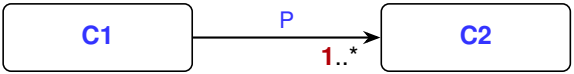
- We have used R to denote either an object property P or the inverse P^- of an object property.
- We have listed the axioms involving object properties, but OWL 2 QL allows for analogous axioms involving data properties.

OWL 2 QL – Mandatory and optional participation to an object property

Mandatory participation

```
C1 rdfs:subClassOf
   [ rdf:type owl:Restriction ;
     owl:onProperty P ;
     owl:someValuesFrom owl:Thing .
   ] .
```

$$C_1 \sqsubseteq \exists P$$



When class C_1 is declared to have a **mandatory participation** to object property P, it means that for every instance o_1 of C_1 there must exist an object o_2 such that the pair (o_1, o_2) is an instance of P. Said differently, C_1 is a subclass of the projection of P on its first component.

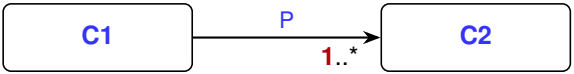
Note: In the graphical notation, the mandatory participation is indicated by a minimum cardinality of 1 associated to the object property. Instead, when the minimum cardinality is 0, the property is **optional** for the instances of the class.
Recall that in the graphical notation, the default cardinality is 0..*, hence we have a mandatory participation only when the cardinality is specified explicitly in the diagram. This is as in OWL 2 QL, where mandatory participation needs to be asserted explicitly through an axiom.

OWL 2 QL – Mandatory and optional participation to an object property

Mandatory participation

```
C1 rdfs:subClassOf
   [ rdf:type owl:Restriction ;
     owl:onProperty P ;
     owl:someValuesFrom owl:Thing .
   ] .
```

$$C_1 \sqsubseteq \exists P$$



When class C_1 is declared to have a **mandatory participation** to object property P, it means that for every instance o_1 of C_1 there must exist an object o_2 such that the pair (o_1, o_2) is an instance of P. Said differently, C_1 is a subclass of the projection of P on its first component.

Note: In the graphical notation, the mandatory participation is indicated by a minimum cardinality of 1 associated to the object property. Instead, when the minimum cardinality is 0, the property is **optional** for the instances of the class.
Recall that in the graphical notation, the default cardinality is 0..*, hence we have a mandatory participation only when the cardinality is specified explicitly in the diagram. This is as in OWL 2 QL, where mandatory participation needs to be asserted explicitly through an axiom.

OWL 2 QL – Mandatory participation – Example

Example:

```
:SeriesActor rdfs:subClassOf
    [ rdf:type owl:Restriction ;
      owl:onProperty :playsIn ;
      owl:someValuesFrom owl:Thing . ] .
```

Inference:

```
<person/5> rdf:type :SeriesActor .
```

$\Longrightarrow$

```
<person/5> rdf:type
    [ rdf:type owl:Restriction ;
      owl:onProperty :playsIn ;
      owl:someValuesFrom owl:Thing . ] .
```

In DL notation: $\mathsf{SeriesActor} \sqsubseteq \exists \mathsf{playsIn}$

$\mathsf{SeriesActor}(\mathit{person/5}) \implies \mathsf{playsIn}(\mathit{person/5}, s)$, for some s

OWL 2 QL – Mandatory participation – Example

Example:

```
:SeriesActor rdfs:subClassOf
    [ rdf:type owl:Restriction ;
      owl:onProperty :playsIn ;
      owl:someValuesFrom owl:Thing . ] .
```

Inference: `<person/5> rdf:type :SeriesActor .`

$\Longrightarrow$

```
<person/5> rdf:type
    [ rdf:type owl:Restriction ;
      owl:onProperty :playsIn ;
      owl:someValuesFrom owl:Thing . ] .
```

In DL notation: $\mathsf{SeriesActor} \sqsubseteq \exists \mathsf{playsIn}$

$\mathsf{SeriesActor}(\mathit{person/5}) \implies \mathsf{playsIn}(\mathit{person/5}, s)$, for some s

OWL 2 QL – Mandatory participation – Example

Example:

```
:SeriesActor rdfs:subClassOf
    [ rdf:type owl:Restriction ;
      owl:onProperty :playsIn ;
      owl:someValuesFrom owl:Thing . ] .
```

Inference: `<person/5> rdf:type :SeriesActor .`

$\Longrightarrow$

```
<person/5> rdf:type
    [ rdf:type owl:Restriction ;
      owl:onProperty :playsIn ;
      owl:someValuesFrom owl:Thing . ] .
```

In DL notation: $\mathsf{SeriesActor} \sqsubseteq \exists \mathsf{playsIn}$

$\mathsf{SeriesActor}(\mathit{person/5}) \implies \mathsf{playsIn}(\mathit{person/5}, s)$, for some s

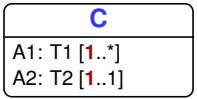
OWL 2 QL – Mandatory and optional data property

A mandatory participation to a data property can be expressed in OWL 2 QL in the same way as for an object property, except that we use `rdfs:Literal` instead of `owl:Thing`. In the graphical notation, it is indicated by a minimum cardinality of 1 associated to the data property.

Mandatory data property

```
C rdfs:subClassOf
  [ rdf:type owl:Restriction ;
    owl:onProperty A ;
    owl:someValuesFrom rdfs:Literal .
  ] .
```

$$C \sqsubseteq \exists A$$



Note: For data properties, in the graphical notation the default cardinality is [1..1], hence in a diagram data properties are mandatory by default.
Instead, when the minimum cardinality is 0, the data property is **optional**, which is the default in OWL 2 QL.

Syntax and semantics of OWL 2 QL KBs

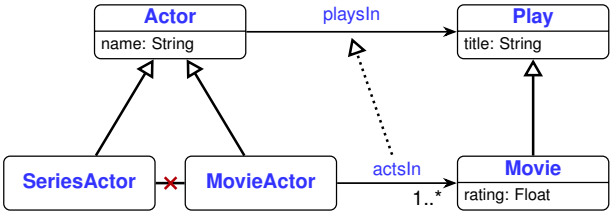
Axiom type	OWL Syntax	DL Syntax	Semantics
Membership (class)	`<a> rdf:type <C>`	$C(a)$	$a \in C^{\mathcal{I}}$
Membership (data property)	`<a> :A <l>`	$A(a, \ell)$	$(a, \ell) \in A^{\mathcal{I}}$
Membership (object property)	`<a1> :P <a2>`	$P(a_1, a_2)$	$(a_1, a_2) \in P^{\mathcal{I}}$
Subclass assertion	`C1 rdfs:subClassOf C2`	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Class disjointness	`C1 owl:disjointWith C2`	$C_1 \sqsubseteq \neg C_2$	$C_1^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} - C_2^{\mathcal{I}}$
Property disjointness	`P1 owl:propertyDisjointWith P2`	$P_1 \sqsubseteq \neg P_2$	$P_1^{\mathcal{I}} \subseteq (\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) - P_2^{\mathcal{I}}$
Domain of a property	`P rdfs:domain C1`	$\exists P \sqsubseteq C_1$	$\{x \mid \exists y.(x, y) \in P^{\mathcal{I}}\} \subseteq C_1^{\mathcal{I}}$
Range of a property	`P rdfs:range C2`	$\exists P^- \sqsubseteq C_2$	$\{y \mid \exists x.(x, y) \in P^{\mathcal{I}}\} \subseteq C_2^{\mathcal{I}}$
Mandatory participation	using `owl:someValuesFrom`	$C \sqsubseteq \exists R$	$C^{\mathcal{I}} \subseteq \exists R^{\mathcal{I}}$
Subproperty assertion	`P1 rdfs:subPropertyOf R2`	$P_1 \sqsubseteq R_2$	$P_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
Inverse property	`P2 owl:inverseOf P1`	$P_1 \equiv P_2^-$	$P_1^{\mathcal{I}} = \{(y, x) \mid (x, y) \in P_2^{\mathcal{I}}\}$

- We have used R to denote either an object property P or the inverse P^- of an object property.
- We have listed the axioms involving object properties, but OWL 2 QL allows for analogous axioms involving data properties.

Representing OWL 2 QL ontologies as UML class diagrams/ER schemas

There is a close correspondence between OWL 2 QL and conceptual modeling formalisms, such as UML class diagrams and ER schemas [Lenzerini & Nobili 1990; Bergamaschi & Sartori 1992; Borgida 1995; C., Lenzerini, et al. 1999; Borgida & Brachman 2003; Berardi et al. 2005; Queralt et al. 2012].

$\mathsf{SeriesActor} \sqsubseteq \mathsf{Actor}$	`rdfs:subClassOf`	subclass
$\mathsf{SeriesActor} \sqsubseteq \neg\mathsf{MovieActor}$	`owl:disjointWith`	disjointness
$\exists\mathsf{playsIn} \sqsubseteq \mathsf{Actor}$	`rdfs:domain`	domain
$\exists\mathsf{playsIn}^{-} \sqsubseteq \mathsf{Play}$	`rdfs:range`	range
$\mathsf{MovieActor} \sqsubseteq \exists\mathsf{actsIn}$	`owl:someValuesFrom`	mandatory participation
$\mathsf{actsIn} \sqsubseteq \mathsf{playsIn}$	`rdfs:subPropertyOf`	sub-association
...		

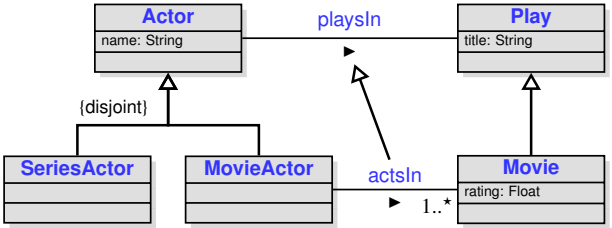


In fact, to visualize an OWL 2 QL ontology, we could have used standard UML class diagrams, instead of the specific graphical notation that we have introduced.

Representing OWL 2 QL ontologies as UML class diagrams/ER schemas

There is a close correspondence between OWL 2 QL and conceptual modeling formalisms, such as UML class diagrams and ER schemas [Lenzerini & Nobili 1990; Bergamaschi & Sartori 1992; Borgida 1995; C., Lenzerini, et al. 1999; Borgida & Brachman 2003; Berardi et al. 2005; Queralt et al. 2012].

$\mathsf{SeriesActor} \sqsubseteq \mathsf{Actor}$	`rdfs:subClassOf`	subclass
$\mathsf{SeriesActor} \sqsubseteq \neg\mathsf{MovieActor}$	`owl:disjointWith`	disjointness
$\exists\mathsf{playsIn} \sqsubseteq \mathsf{Actor}$	`rdfs:domain`	domain
$\exists\mathsf{playsIn}^{-} \sqsubseteq \mathsf{Play}$	`rdfs:range`	range
$\mathsf{MovieActor} \sqsubseteq \exists\mathsf{actsIn}$	`owl:someValuesFrom`	mandatory participation
$\mathsf{actsIn} \sqsubseteq \mathsf{playsIn}$	`rdfs:subPropertyOf`	sub-association
$\ldots$		



In fact, to visualize an OWL 2 QL ontology, we could have used standard UML class diagrams, instead of the specific graphical notation that we have introduced.

Outline

Query answering – Which query language to use

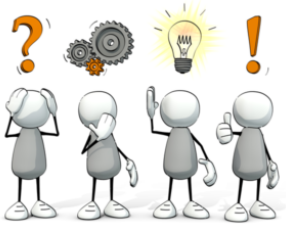
Querying under incomplete information

Query answering is not simply query evaluation, but a form of logical inference, and requires reasoning.

Query answering – Which query language to use

Querying under incomplete information

Query answering is not simply query evaluation, but a form of logical inference, and requires reasoning.



Two borderline cases for choosing the language for querying KBs:

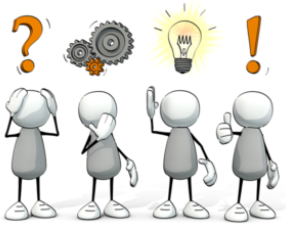
1. Use the **ontology language** as query language.

2. Use **Full SQL** (or equivalently, first-order logic).

Query answering – Which query language to use

Querying under incomplete information

Query answering is not simply query evaluation, but a form of logical inference, and requires reasoning.



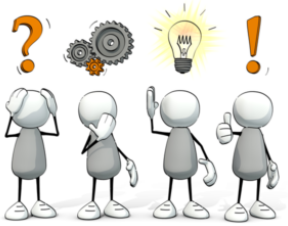
Two borderline cases for choosing the language for querying KBs:

1. Use the **ontology language** as query language.
   - Ontology languages are tailored for capturing intensional relationships.
   - They are quite poor as query languages.
2. Use **Full SQL** (or equivalently, first-order logic).

Query answering – Which query language to use

Querying under incomplete information

Query answering is not simply query evaluation, but a form of logical inference, and requires reasoning.



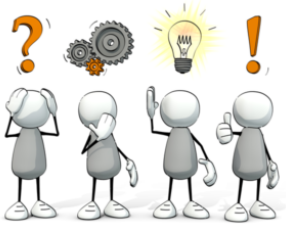
Two borderline cases for choosing the language for querying KBs:

1. Use the **ontology language** as query language.
   - Ontology languages are tailored for capturing intensional relationships.
   - They are quite poor as query languages.
2. Use **Full SQL** (or equivalently, first-order logic).
   - Problem: in a setting with incomplete information, query answering is undecidable (FOL validity).

Query answering – Which query language to use

Querying under incomplete information

Query answering is not simply query evaluation, but a form of logical inference, and requires reasoning.



Two borderline cases for choosing the language for querying KBs:

1. Use the **ontology language** as query language.
   - Ontology languages are tailored for capturing intensional relationships.
   - They are quite poor as query languages.
2. Use **Full SQL** (or equivalently, first-order logic).
   - Problem: in a setting with incomplete information, query answering is undecidable (FOL validity).

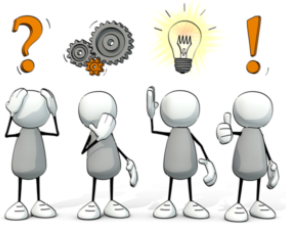
Conjunctive queries

A good tradeoff is to use conjunctive queries (CQs) or unions of CQs (UCQs), corresponding to SQL/relational algebra (union) select-project-join queries.

Query answering – Which query language to use

Querying under incomplete information

Query answering is not simply query evaluation, but a form of logical inference, and requires reasoning.



Two borderline cases for choosing the language for querying KBs:

1. Use the **ontology language** as query language.
   - Ontology languages are tailored for capturing intensional relationships.
   - They are quite poor as query languages.
2. Use **Full SQL** (or equivalently, first-order logic).
   - Problem: in a setting with incomplete information, query answering is undecidable (FOL validity).

Conjunctive queries – Are concretely represented in **SPARQL**

A good tradeoff is to use conjunctive queries (CQs) or unions of CQs (UCQs), corresponding to SQL/relational algebra (union) select-project-join queries.

SPARQL query language

- Is the standard query language for RDF data. [W3C Rec. 2008, 2013]
- Core query mechanism is based on **graph matching**.

```
SELECT ?p ?t
WHERE { ?p rdf:type Professor .
        ?p teaches ?c .
        ?c rdf:type Course .
        ?c title ?t .
      }
```

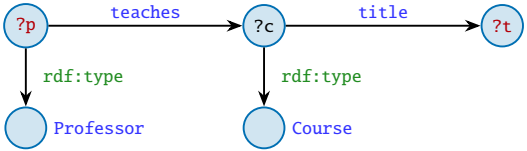

Additional language features (SPARQL 1.1):

- `UNION`: matches one of alternative graph patterns
- `OPTIONAL`: produces a match even when part of the pattern is missing
- complex `FILTER` conditions
- `GROUP BY`, to express aggregations
- `MINUS`, to remove possible solutions
- property paths (regular expressions)
- ...

SPARQL query language

- Is the standard query language for RDF data. [W3C Rec. 2008, 2013]
- Core query mechanism is based on **graph matching**.

```
SELECT ?p ?t
WHERE { ?p rdf:type Professor .
        ?p teaches ?c .
        ?c rdf:type Course .
        ?c title ?t .
      }
```



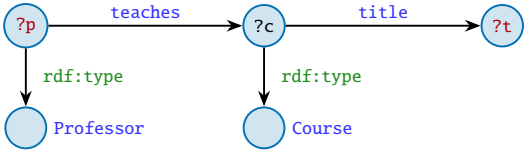
Additional language features (SPARQL 1.1):

- `UNION`: matches one of alternative graph patterns
- `OPTIONAL`: produces a match even when part of the pattern is missing
- complex `FILTER` conditions
- `GROUP BY`, to express aggregations
- `MINUS`, to remove possible solutions
- property paths (regular expressions)
- ...

SPARQL query language

- Is the standard query language for RDF data. [W3C Rec. 2008, 2013]
- Core query mechanism is based on **graph matching**.

```
SELECT ?p ?t
WHERE { ?p rdf:type Professor .
        ?p teaches ?c .
        ?c rdf:type Course .
        ?c title ?t .
      }
```



Additional language features (SPARQL 1.1):

- `UNION`: matches one of alternative graph patterns
- `OPTIONAL`: produces a match even when part of the pattern is missing
- complex `FILTER` conditions
- `GROUP BY`, to express aggregations
- `MINUS`, to remove possible solutions
- property paths (regular expressions)
- ...

SPARQL Basic Graph Patterns

Basic Graph Pattern (BGP) are the simplest form of SPARQL query, asking for a pattern in the RDF graph, made up of triple patterns.

Example: BGP

```
SELECT ?p ?ln ?c ?t
WHERE {
  ?p :lastName ?ln .
  ?p :teaches ?c .
  ?c :title ?t .
}
```

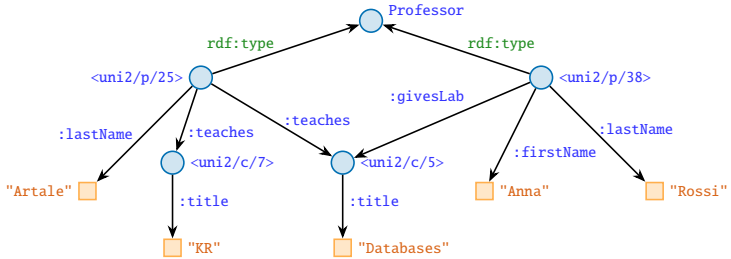
SPARQL Basic Graph Patterns

Basic Graph Pattern (BGP) are the simplest form of SPARQL query, asking for a pattern in the RDF graph, made up of triple patterns.

Example: BGP

```
SELECT ?p ?ln ?c ?t
WHERE {
  ?p :lastName ?ln .
  ?p :teaches ?c .
  ?c :title ?t .
}
```

When evaluated over the RDF graph



. . . the query returns:

p	ln	c	t
<uni2/p/25>	"Artale"	<uni2/c/5>	"Databases"
<uni2/p/25>	"Artale"	<uni2/c/7>	"KR"

Abbreviated syntax for Basic Graph Patterns

We can use an abbreviated syntax for BGPs, that avoids repeating the subject of triple patterns.

Example: BGP

```
SELECT ?p ?ln ?c ?t ?r
WHERE {
  ?p :lastName ?ln .
  ?p :teaches ?c .
  ?c :title ?t .
  ?c :room ?r .
}
```

Example: BGP with abbreviated syntax

```
SELECT ?p ?ln ?c ?t ?r
WHERE {
  ?p :lastName ?ln ;
     :teaches ?c .
  ?c :title ?t ;
     :room ?r .
}
```

When we end a triple pattern with a ';' (instead of '.'), the next triple pattern uses the same subject (which therefore is not repeated).

Abbreviated syntax for Basic Graph Patterns

We can use an abbreviated syntax for BGPs, that avoids repeating the subject of triple patterns.

Example: BGP

```
SELECT ?p ?ln ?c ?t ?r
WHERE {
  ?p :lastName ?ln .
  ?p :teaches ?c .
  ?c :title ?t .
  ?c :room ?r .
}
```

Example: BGP with abbreviated syntax

```
SELECT ?p ?ln ?c ?t ?r
WHERE {
  ?p :lastName ?ln ;
     :teaches ?c .
  ?c :title ?t ;
     :room ?r .
}
```

When we end a triple pattern with a '`;`' (instead of '`.`'), the next triple pattern uses the same subject (which therefore is not repeated).

Projecting out variables in a SPARQL query

A query may also return only a subset of the variables used in the BGP.

Example: BGP with projection

```
SELECT ?ln ?t
WHERE {
  ?p :lastName ?ln .
  ?p :teaches ?c .
  ?c :title ?t .
}
```

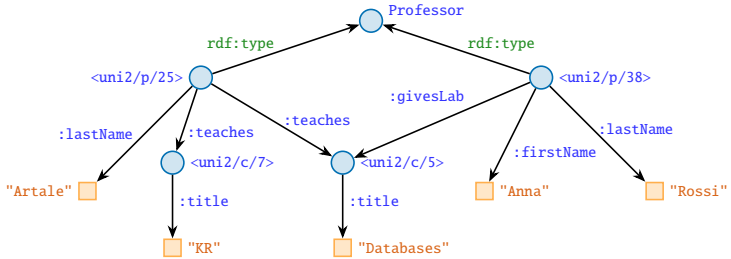
Projecting out variables in a SPARQL query

A query may also return only a subset of the variables used in the BGP.

Example: BGP with projection

```
SELECT ?ln ?t
WHERE {
  ?p :lastName ?ln .
  ?p :teaches ?c .
  ?c :title ?t .
}
```

When evaluated over the RDF graph



. . . the query returns:

ln	t
"Artale"	"Databases"
"Artale"	"KR"

Anonymous variables

We can use `[...]` to represent an anonymous variable.

Example: BGP

```
SELECT ?ln ?t ?r
WHERE {
  ?p :lastName ?ln ;
     :teaches ?c .
  ?c :title ?t ;
     :room ?r .
}
```

Example: BGP with anonymous variable

```
SELECT ?ln ?t ?r
WHERE {
  ?p :lastName ?ln ;
     :teaches
      [ :title ?t ;
        :room ?r .  ] .
}
```

Within the square brackets, the triple patterns, separated by ';', all have the anonymous variable as subject.

Anonymous variables

We can use `[...]` to represent an anonymous variable.

Example: BGP

```
SELECT ?ln ?t ?r
WHERE {
  ?p :lastName ?ln ;
     :teaches ?c .
  ?c :title ?t ;
     :room ?r .
}
```

Example: BGP with anonymous variable

```
SELECT ?ln ?t ?r
WHERE {
  ?p :lastName ?ln ;
     :teaches
   [ :title ?t ;
     :room ?r .  ] .
}
```

Within the square brackets, the triple patterns, separated by ';', all have the anonymous variable as subject.

Union of Basic Graph Patterns

Example: BGPs with UNION

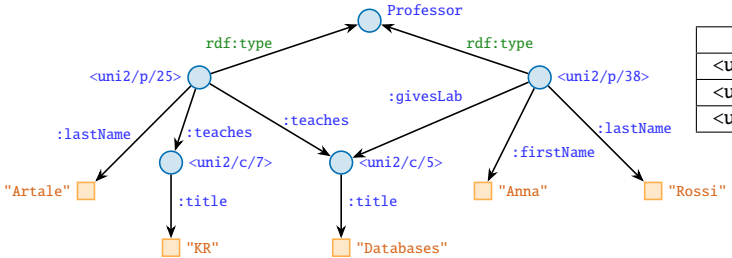
```
SELECT ?p ?ln ?c
WHERE {
  { ?p :lastName ?ln .    ?p :teaches ?c . }
  UNION
  { ?p :lastName ?ln .    ?p :givesLab ?c . }
}
```

Union of Basic Graph Patterns

Example: BGPs with UNION

```
SELECT ?p ?ln ?c
WHERE {
  { ?p :lastName ?ln .    ?p :teaches ?c . }
  UNION
  { ?p :lastName ?ln .    ?p :givesLab ?c . }
}
```

When evaluated over



. . . the query returns:

p	ln	c
<uni2/p/25>	"Artale"	<uni2/c/5>
<uni2/p/25>	"Artale"	<uni2/c/7>
<uni2/p/38>	"Rossi"	<uni2/c/5>

Extending BGPs with OPTIONAL

We might want to add information when available, but **not reject** a solution **when some part of the query does not match**.

Example: BGP with OPTIONAL

```
SELECT ?p ?fn ?ln
WHERE {
  ?p :lastName ?ln .
  OPTIONAL {
    ?p :firstName ?fn .
  }
}
```

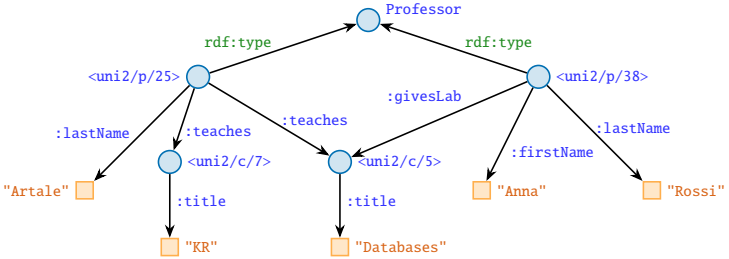
Extending BGPs with OPTIONAL

We might want to add information when available, but **not reject** a solution **when some part of the query does not match**.

Example: BGP with OPTIONAL

```
SELECT ?p ?fn ?ln
WHERE {
  ?p :lastName ?ln .
  OPTIONAL {
    ?p :firstName ?fn .
  }
}
```

When evaluated over the RDF graph



. . . the query returns:

p	fn	ln
<uni2/p/25>		"Artale"
<uni2/p/38>	"Anna"	"Rossi"

ORDER BY, LIMIT, and OFFSET

We might be interested in obtaining the results in a certain order, and/or only some of the results. This is controlled by three clauses, appended to the `WHERE {}` block: `ORDER BY`, `LIMIT`, and `OFFSET`.

Example: Ordering and limiting the results

```
SELECT ?ln ?t ?r
WHERE {
  ?p :lastName ?ln ;
     :teaches ?c .
  ?c :title ?t ;  :room ?r .
}
ORDER BY ?ln
LIMIT 10
OFFSET 5
```

Example: Multiple order comparators

```
SELECT ?ln ?t ?r
WHERE {
  ?p :lastName ?ln ;
     :teaches ?c .
  ?c :title ?t ;  :room ?r .
}
ORDER BY ASC(?ln) DESC(?t)
```

The default is no limit, and offset 0.

Each order comparator consists of an expression, with an optional order modifier applied to it:

- `ASC()` for ascending order, which is the default;
- `DESC()` for descending order.

ORDER BY, LIMIT, and OFFSET

We might be interested in obtaining the results in a certain order, and/or only some of the results. This is controlled by three clauses, appended to the `WHERE {}` block: `ORDER BY`, `LIMIT`, and `OFFSET`.

Example: Ordering and limiting the results

```
SELECT ?ln ?t ?r
WHERE {
  ?p :lastName ?ln ;
     :teaches ?c .
  ?c :title ?t ;  :room ?r .
}
ORDER BY ?ln
LIMIT 10
OFFSET 5
```

Example: Multiple order comparators

```
SELECT ?ln ?t ?r
WHERE {
  ?p :lastName ?ln ;
     :teaches ?c .
  ?c :title ?t ;  :room ?r .
}
ORDER BY ASC(?ln) DESC(?t)
```

The default is no limit, and offset 0.

Each order comparator consists of an expression, with an optional order modifier applied to it:

- `ASC()` for ascending order, which is the default;
- `DESC()` for descending order.

ORDER BY, LIMIT, and OFFSET

We might be interested in obtaining the results in a certain order, and/or only some of the results. This is controlled by three clauses, appended to the `WHERE {}` block: `ORDER BY`, `LIMIT`, and `OFFSET`.

Example: Ordering and limiting the results

```
SELECT ?ln ?t ?r
WHERE {
  ?p :lastName ?ln ;
     :teaches ?c .
  ?c :title ?t ;  :room ?r .
}
ORDER BY ?ln
LIMIT 10
OFFSET 5
```

Example: Multiple order comparators

```
SELECT ?ln ?t ?r
WHERE {
  ?p :lastName ?ln ;
     :teaches ?c .
  ?c :title ?t ;  :room ?r .
}
ORDER BY ASC(?ln) DESC(?t)
```

The default is no limit, and offset 0.

Each order comparator consists of an expression, with an optional order modifier applied to it:

- `ASC()` for ascending order, which is the default;
- `DESC()` for descending order.

FILTER conditions

We might want to select only those answers to a query that respect some condition.
This can be achieved by adding to the query one or more FILTER conditions.

Example: BGP with a FILTER condition

```
SELECT ?ln ?dob
WHERE {
  ?p :lastName ?ln ;  :isBorn ?dob .
  FILTER("1990-01-01"^^xsd:dateTime <= ?dob   &&
         ?dob < "1996-01-01"^^xsd:dateTime) .
}
```

More in general, the argument of FILTER() is an expression returning an xsd:boolean, built using:

- comparison atoms, which use the comparison operators: =, !=, <, >, <=, >=;
- logical connectives: && and ||;
- EXISTS { *pattern* } and NOT EXISTS { *pattern* }, where *pattern* is a graph pattern;
- SPARQL functions (for more details, see the document defining the SPARQL standard).

FILTER conditions

We might want to select only those answers to a query that respect some condition.
This can be achieved by adding to the query one or more FILTER conditions.

Example: BGP with a FILTER condition

```
SELECT ?ln ?dob
WHERE {
  ?p :lastName ?ln ;  :isBorn ?dob .
  FILTER("1990-01-01"^^xsd:dateTime <= ?dob   &&
         ?dob < "1996-01-01"^^xsd:dateTime) .
}
```

More in general, the argument of FILTER() is an expression returning an xsd:boolean, built using:

- comparison atoms, which use the comparison operators: =, !=, <, >, <=, >=;
- logical connectives: && and ||;
- EXISTS { *pattern* } and NOT EXISTS { *pattern* }, where *pattern* is a graph pattern;
- SPARQL functions (for more details, see the document defining the SPARQL standard).

FILTER conditions

We might want to select only those answers to a query that respect some condition.
This can be achieved by adding to the query one or more FILTER conditions.

Example: BGP with a FILTER condition

```
SELECT ?ln ?dob
WHERE {
  ?p :lastName ?ln ;  :isBorn ?dob .
  FILTER("1990-01-01"^^xsd:dateTime <= ?dob   &&
         ?dob < "1996-01-01"^^xsd:dateTime) .
}
```

More in general, the argument of FILTER() is an expression returning an xsd:boolean, built using:

- comparison atoms, which use the comparison operators: =, !=, <, >, <=, >=;
- logical connectives: && and ||;
- EXISTS { *pattern* } and NOT EXISTS { *pattern* }, where *pattern* is a graph pattern;
- SPARQL functions (for more details, see the document defining the SPARQL standard).

SPARQL algebra

We have seen the following features of the SPARQL algebra:

- Basic Graph Patterns
- `UNION`
- `OPTIONAL`
- `ORDER BY`, `LIMIT`, `OFFSET`
- `FILTER` conditions

The overall algebra has additional features:

- `GROUP BY`, to express aggregations and support aggregation operators
- `MINUS`, to remove possible solutions
- path expressions, corresponding to regular expressions

SPARQL algebra

We have seen the following features of the SPARQL algebra:

- Basic Graph Patterns
- `UNION`
- `OPTIONAL`
- `ORDER BY`, `LIMIT`, `OFFSET`
- `FILTER` conditions

The overall algebra has additional features:

- `GROUP BY`, to express aggregations and support aggregation operators
- `MINUS`, to remove possible solutions
- path expressions, corresponding to regular expressions

Outline

Use of mappings

In the VKG framework, the mapping encodes how the data in the sources should be used to create the Virtual Knowledge Graph, which is formulated in the vocabulary of the ontology.

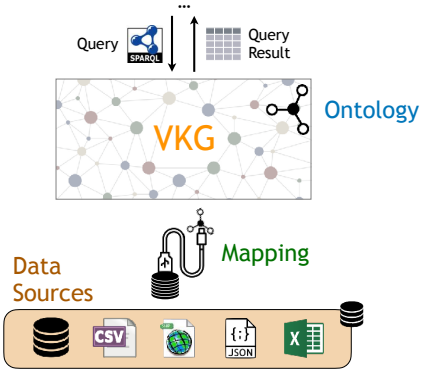
Use of mappings

In the VKG framework, the mapping encodes how the data in the sources should be used to create the Virtual Knowledge Graph, which is formulated in the vocabulary of the ontology.

VKG defined from the mapping and the data.

- Queries are answered with respect to the ontology and the data of the VKG.
- The data of the VKG is not materialized (it is virtual!).
- Instead, the information in the ontology and the mapping is used to translate queries over the ontology into queries formulated over the sources.

Note: The graph is **always up to date** wrt the data sources.

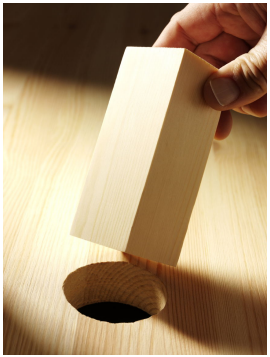


Mismatch between data layer and ontology

Impedance mismatch

- Relational databases store values.
- Knowledge bases / ontologies represent both objects and values.

We need to construct the ontology objects from the database values.



Proposed solution

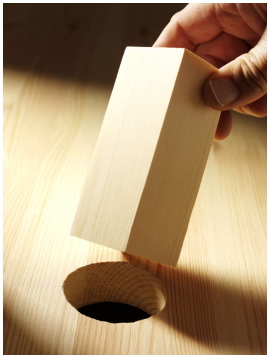
The specification of **how to construct the ontology objects** that populate the virtual knowledge graph from the database values **is embedded in the mapping** between the data sources and the ontology.

Mismatch between data layer and ontology

Impedance mismatch

- Relational databases store values.
- Knowledge bases / ontologies represent both objects and values.

We need to construct the ontology objects from the database values.



Proposed solution

The specification of **how to construct the ontology objects** that populate the virtual knowledge graph from the database values **is embedded in the mapping** between the data sources and the ontology.

VKG mapping

The **mapping** consists of a set of assertions of the form:

$$Q_{sql}(\vec{x}) \rightsquigarrow \Psi(\vec{t}, \vec{x})$$

- $Q_{sql}(\vec{x})$ is the source query expressed in SQL.
- $\Psi(\vec{t}, \vec{x})$ is the target, consisting of a set of **triple patterns** (i.e., atoms) that refer to the classes and properties of the ontology and make use of the answer variables $\vec{x}$ of the SQL query.

To address the impedance mismatch, in the target query:

- we specify how to construct valid IRIs (that act as object identifiers), by concatenating database values and string constants;
- to refer to a database value, we use an answer variable of the source query;
- we call a term that constructs an IRI by referring to answer variables of the source query, an **IRI-template**.

VKG mapping

The **mapping** consists of a set of assertions of the form:

$$Q_{sql}(\vec{x}) \rightsquigarrow \Psi(\vec{t}, \vec{x})$$

- $Q_{sql}(\vec{x})$ is the source query expressed in SQL.
- $\Psi(\vec{t}, \vec{x})$ is the target, consisting of a set of **triple patterns** (i.e., atoms) that refer to the classes and properties of the ontology and make use of the answer variables $\vec{x}$ of the SQL query.

To address the impedance mismatch, in the target query:

- we specify how to construct valid IRIs (that act as object identifiers), by concatenating database values and string constants;
- to refer to a database value, we use an answer variable of the source query;
- we call a term that constructs an IRI by referring to answer variables of the source query, an **IRI-template**.

Triple patterns and IRI-templates

Intuition behind the mapping

The answers returned by the SQL query in the source-part of the mapping are used to create, via the IRI-templates, the objects (and values) that populate the classes / properties in the target part.

More precisely:

- Each **triple pattern** in the target part has one of the forms:

$$\begin{array}{ll} \mathbf{iri}_1(\vec{x}_1) \ \texttt{rdf:type} \ C & \text{where } C \text{ is a class of the ontology, or} \\ \mathbf{iri}_1(\vec{x}_1) \ \mathit{prop} \ \mathbf{iri}_2(\vec{x}_2) & \text{where } \mathit{prop} \text{ is a (data or object) property of the ontology.} \end{array}$$

- For each answer tuple $\vec{a}$ returned by the source query $Q_{sql}(\vec{x})$ (when evaluated over the database), the **iri-template** $\mathbf{iri}_i(\vec{x}_i)$ generates an object / value $\mathbf{iri}_i(\vec{a}_i)$ of the VKG.
- Such objects / values are then used to populate the classes and properties of the ontology according to what specified in the target part of the mapping.

In this way we provide a solution to the impedance mismatch problem.

Triple patterns and IRI-templates

Intuition behind the mapping

The answers returned by the SQL query in the source-part of the mapping are used to create, via the IRI-templates, the objects (and values) that populate the classes / properties in the target part.

More precisely:

- Each **triple pattern** in the target part has one of the forms:

$$\mathbf{iri}_1(\vec{x}_1)\ \ \texttt{rdf:type}\ C \qquad \text{where } C \text{ is a class of the ontology, or}$$
$$\mathbf{iri}_1(\vec{x}_1)\ \ prop\ \ \mathbf{iri}_2(\vec{x}_2) \qquad \text{where } prop \text{ is a (data or object) property of the ontology.}$$

- For each answer tuple $\vec{a}$ returned by the source query $Q_{sql}(\vec{x})$ (when evaluated over the database), the **iri-template** $\mathbf{iri}_i(\vec{x}_i)$ generates an object / value $\mathbf{iri}_i(\vec{a}_i)$ of the VKG.
- Such objects / values are then used to populate the classes and properties of the ontology according to what specified in the target part of the mapping.

In this way we provide a solution to the impedance mismatch problem.

A concrete mapping language

We describe the concrete mapping language adopted by the *Ontop* system.

In the *Ontop* mapping language, each mapping assertion is made up of three parts:

- A **mapping identifier**, which is convenient to refer to a specific mapping.
- The **source part**, which is a regular SQL query over the data source(s).
- The **target part**, which is a set of triple patterns that make use of IRI-templates. In the target part, the answer variables of the source part are enclosed in `{...}`.

Mapping m_1

- Mapping identifier: `m1`
- Source part:

```
SELECT mcode, mtitle
FROM MOVIE
WHERE type = "m"
```

- Target part:

```
:m/{mcode} rdf:type :Movie .
:m/{mcode} :title {mtitle} .
```

Mapping m_2

- Mapping identifier: `m2`
- Source part:

```
SELECT M.mcode, A.acode
FROM MOVIE M, ACTOR A
WHERE M.mcode = A.pcode
  AND M.type = "m"
```

- Target part:

```
:a/{acode} :actsIn :m/{mcode} .
```

A concrete mapping language

We describe the concrete mapping language adopted by the *Ontop* system.

In the *Ontop* mapping language, each mapping assertion is made up of three parts:

- A **mapping identifier**, which is convenient to refer to a specific mapping.
- The **source part**, which is a regular SQL query over the data source(s).
- The **target part**, which is a set of triple patterns that make use of IRI-templates. In the target part, the answer variables of the source part are enclosed in `{...}`.

Mapping m_1

- Mapping identifier: `m1`
- Source part:

```
SELECT mcode, mtitle
FROM MOVIE
WHERE type = "m"
```

- Target part:

```
:m/{mcode} rdf:type :Movie .
:m/{mcode} :title {mtitle} .
```

Mapping m_2

- Mapping identifier: `m2`
- Source part:

```
SELECT M.mcode, A.acode
FROM MOVIE M, ACTOR A
WHERE M.mcode = A.pcode
  AND M.type = "m"
```

- Target part:

```
:a/{acode} :actsIn :m/{mcode} .
```

A concrete mapping language

We describe the concrete mapping language adopted by the *Ontop* system.

In the *Ontop* mapping language, each mapping assertion is made up of three parts:

- A **mapping identifier**, which is convenient to refer to a specific mapping.
- The **source part**, which is a regular SQL query over the data source(s).
- The **target part**, which is a set of triple patterns that make use of IRI-templates. In the target part, the answer variables of the source part are enclosed in `{...}`.

Mapping m_1

- Mapping identifier: `m1`
- Source part:

```
SELECT mcode, mtitle
FROM MOVIE
WHERE type = "m"
```

- Target part:

```
:m/{mcode} rdf:type :Movie .
:m/{mcode} :title {mtitle} .
```

Mapping m_2

- Mapping identifier: `m2`
- Source part:

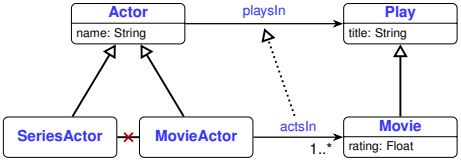
```
SELECT M.mcode, A.acode
FROM MOVIE M, ACTOR A
WHERE M.mcode = A.pcode
  AND M.type = "m"
```

- Target part:

```
:a/{acode} :actsIn :m/{mcode} .
```

Mapping language – Example

Ontology $\mathcal{O}$:



Mapping $\mathcal{M}$:

m_1: `SELECT mcode, mtitle FROM MOVIE`
`WHERE type = "m"`
⇝ `:m/{mcode} rdf:type :Movie .`
`:m/{mcode} :title {mtitle} .`

m_2: `SELECT M.mcode, A.acode FROM MOVIE M, ACTOR A`
`WHERE M.mcode = A.pcode AND M.type = "m"`
⇝ `:a/{acode} :actsIn :m/{mcode} .`

Database $\mathcal{D}$:

MOVIE

mcode	*mtitle*	*myear*	*type*	···
5118	The Matrix	1999	m	···
8234	Altered Carbon	2018	s	···
2281	Blade Runner	1982	m	···

ACTOR

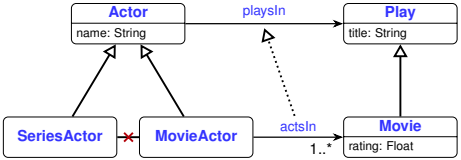
pcode	*acode*	*aname*	···
5118	438	K. Reeves	···
5118	572	C.A. Moss	···
2281	271	H. Ford	···

The mapping $\mathcal{M}$ applied to database $\mathcal{D}$ generates the virtual knowledge graph $\mathcal{M}(\mathcal{D})$:

`:m/5118 rdf:type :Movie .` `:m/5118 :title "The Matrix" .`
`:m/2281 rdf:type :Movie .` `:m/2281 :title "Blade Runner" .`
`:a/438 :actsIn :m/5118 .` `:a/572 :actsIn :m/5118 .` `:a/271 :actsIn :m/2281 .`

Mapping language – Example

Ontology $\mathcal{O}$:



Mapping $\mathcal{M}$:

m_1: **SELECT** mcode, mtitle **FROM** MOVIE
WHERE type = "m"
⇝ :m/{mcode} **rdf:type** :Movie .
:m/{mcode} :title {mtitle} .

m_2: **SELECT** M.mcode, A.acode **FROM** MOVIE M, ACTOR A
WHERE M.mcode = A.pcode **AND** M.type = "m"
⇝ :a/{acode} :actsIn :m/{mcode} .

Database $\mathcal{D}$:

MOVIE

mcode	*mtitle*	*myear*	*type*	···
5118	The Matrix	1999	m	···
8234	Altered Carbon	2018	s	···
2281	Blade Runner	1982	m	···

ACTOR

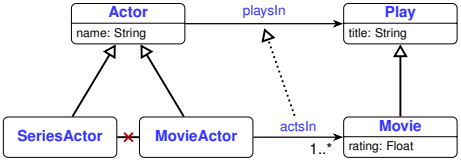
pcode	*acode*	*aname*	···
5118	438	K. Reeves	···
5118	572	C.A. Moss	···
2281	271	H. Ford	···

The mapping $\mathcal{M}$ applied to database $\mathcal{D}$ generates the virtual knowledge graph $\mathcal{M}(\mathcal{D})$:

:m/5118 **rdf:type** :Movie . :m/5118 :title "The Matrix" .
:m/2281 **rdf:type** :Movie . :m/2281 :title "Blade Runner" .
:a/438 :actsIn :m/5118 . :a/572 :actsIn :m/5118 . :a/271 :actsIn :m/2281 .

Mapping language – Example

Ontology $\mathcal{O}$:



Mapping $\mathcal{M}$:

m_1: **SELECT** mcode, mtitle **FROM** MOVIE
WHERE type = "m"
⇝ :m/{mcode} **rdf:type** :Movie .
:m/{mcode} :title {mtitle} .

m_2: **SELECT** M.mcode, A.acode **FROM** MOVIE M, ACTOR A
WHERE M.mcode = A.pcode **AND** M.type = "m"
⇝ :a/{acode} :actsIn :m/{mcode} .

Database $\mathcal{D}$:

MOVIE

mcode	*mtitle*	*myear*	*type*	···
5118	The Matrix	1999	m	···
8234	Altered Carbon	2018	s	···
2281	Blade Runner	1982	m	···

ACTOR

pcode	*acode*	*aname*	···
5118	438	K. Reeves	···
5118	572	C.A. Moss	···
2281	271	H. Ford	···

The mapping $\mathcal{M}$ applied to database $\mathcal{D}$ generates the virtual knowledge graph $\mathcal{M}(\mathcal{D})$:

:m/5118 **rdf:type** :Movie . :m/5118 :title "The Matrix" .
:m/2281 **rdf:type** :Movie . :m/2281 :title "Blade Runner" .
:a/438 :actsIn :m/5118 . :a/572 :actsIn :m/5118 . :a/271 :actsIn :m/2281 .

Standard mapping languages

Several proposals for concrete languages to map a relational DB to an ontology:

- They assume that the ontology is populated in terms of RDF triples.
- Some template mechanism is used to specify the triples to instantiate.

Examples: D2RQ[1], SML[2], Ontop[3]

R2RML

- Most popular RDB to RDF mapping language
- W3C Recommendation 27 Sep. 2012, `http://www.w3.org/TR/r2rml/`
- R2RML mappings are themselves expressed as RDF graphs and written in Turtle syntax.

[1] `http://d2rq.org/d2rq-language`
[2] `http://sparqlify.org/wiki/Sparqlification_mapping_language`
[3] `https://github.com/ontop/ontop/wiki/ontopOBDAModel#Mapping_axioms`

Standard mapping languages

Several proposals for concrete languages to map a relational DB to an ontology:

- They assume that the ontology is populated in terms of RDF triples.
- Some template mechanism is used to specify the triples to instantiate.

Examples: D2RQ[1], SML[2], Ontop[3]

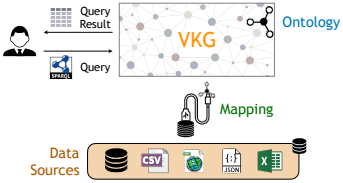
R2RML

- Most popular RDB to RDF mapping language
- W3C Recommendation 27 Sep. 2012, `http://www.w3.org/TR/r2rml/`
- R2RML mappings are themselves expressed as RDF graphs and written in Turtle syntax.

[1] http://d2rq.org/d2rq-language
[2] http://sparqlify.org/wiki/Sparqlification_mapping_language
[3] https://github.com/ontop/ontop/wiki/ontopOBDAModel#Mapping_axioms

Outline

VKGs: Formalization



To formalize VKGs, we distinguish between the intensional and the extensional level information.

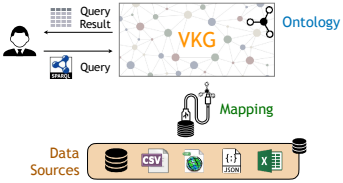
A **VKG specification** is a triple $\mathcal{P} = (\mathcal{O}, \mathcal{M}, \mathcal{S})$, where:

- $\mathcal{O}$ is an ontology (expressed in OWL 2 QL),
- $\mathcal{S}$ is a (possibly federated) relational database schema for the data sources, possibly with integrity constraints,
- $\mathcal{M}$ is a set of (R2RML) mapping assertions between $\mathcal{O}$ and $\mathcal{S}$.

A **VKG instance** is a pair $\mathcal{J} = (\mathcal{P}, \mathcal{D})$, where

- $\mathcal{P} = (\mathcal{O}, \mathcal{M}, \mathcal{S})$ is a VKG specification, and
- $\mathcal{D}$ is a (possibly federated) relational database compliant with $\mathcal{S}$.

VKGs: Formalization



To formalize VKGs, we distinguish between the intensional and the extensional level information.

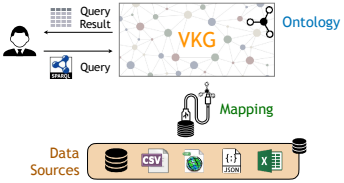
A **VKG specification** is a triple $\mathcal{P} = (\mathcal{O}, \mathcal{M}, \mathcal{S})$, where:

- $\mathcal{O}$ is an ontology (expressed in OWL 2 QL),
- $\mathcal{S}$ is a (possibly federated) relational database schema for the data sources, possibly with integrity constraints,
- $\mathcal{M}$ is a set of (R2RML) mapping assertions between $\mathcal{O}$ and $\mathcal{S}$.

A **VKG instance** is a pair $\mathcal{J} = (\mathcal{P}, \mathcal{D})$, where

- $\mathcal{P} = (\mathcal{O}, \mathcal{M}, \mathcal{S})$ is a VKG specification, and
- $\mathcal{D}$ is a (possibly federated) relational database compliant with $\mathcal{S}$.

VKGs: Formalization



To formalize VKGs, we distinguish between the intensional and the extensional level information.

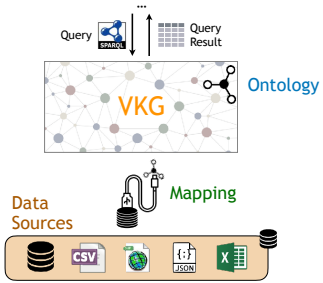
A **VKG specification** is a triple $\mathcal{P} = (\mathcal{O}, \mathcal{M}, \mathcal{S})$, where:

- $\mathcal{O}$ is an ontology (expressed in OWL 2 QL),
- $\mathcal{S}$ is a (possibly federated) relational database schema for the data sources, possibly with integrity constraints,
- $\mathcal{M}$ is a set of (R2RML) mapping assertions between $\mathcal{O}$ and $\mathcal{S}$.

A **VKG instance** is a pair $\mathcal{J} = (\mathcal{P}, \mathcal{D})$, where

- $\mathcal{P} = (\mathcal{O}, \mathcal{M}, \mathcal{S})$ is a VKG specification, and
- $\mathcal{D}$ is a (possibly federated) relational database compliant with $\mathcal{S}$.

Semantics of VKGs



Remember:

- The mapping $\mathcal{M}$ generates from the data $\mathcal{D}$ in the sources a **virtual knowledge graph** $\mathcal{V} = \mathcal{M}(\mathcal{D})$.
- Therefore, the pair $(\mathcal{O}, \mathcal{M}(\mathcal{D}))$ is a knowledge base.
- Semantics for a VKG instance can thus be defined in terms of the semantics of a KB.

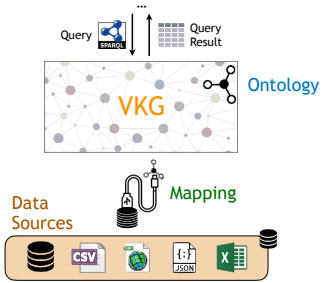
Model of a VKG instance

An interpretation $\mathcal{I}$ is a **model** of $(\mathcal{P}, \mathcal{D})$, denoted as $\mathcal{I} \models (\mathcal{P}, \mathcal{D})$, if $\mathcal{I}$ is a model of the KB $(\mathcal{O}, \mathcal{M}(\mathcal{D}))$.

Note:

- In general, $(\mathcal{P}, \mathcal{D})$ has infinitely many models, and some of these might be infinite.
- However, for query answering, we do not need to compute such models.

Semantics of VKGs



Remember:

- The mapping $\mathcal{M}$ generates from the data $\mathcal{D}$ in the sources a **virtual knowledge graph** $\mathcal{V} = \mathcal{M}(\mathcal{D})$.
- Therefore, the pair $(\mathcal{O}, \mathcal{M}(\mathcal{D}))$ is a knowledge base.
- Semantics for a VKG instance can thus be defined in terms of the semantics of a KB.

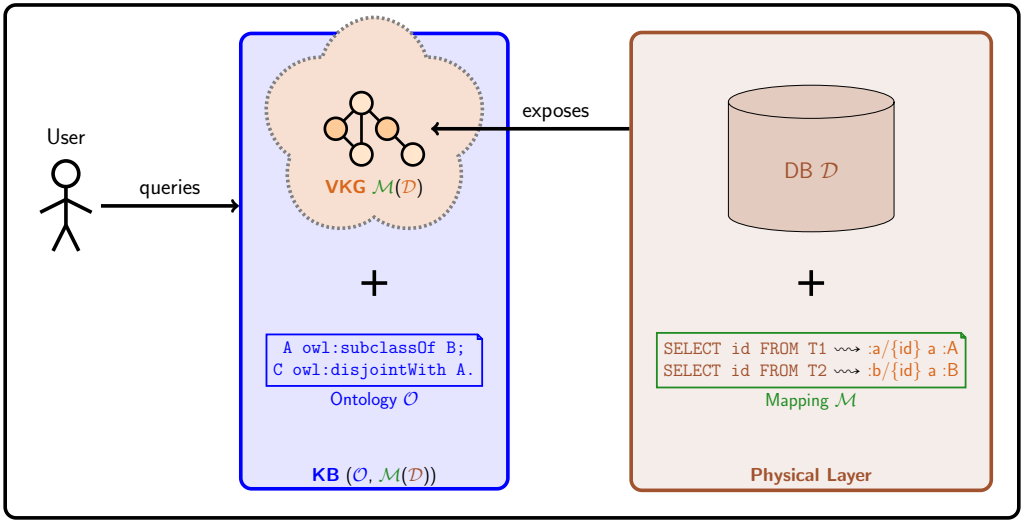
Model of a VKG instance

An interpretation $\mathcal{I}$ is a **model** of $(\mathcal{P}, \mathcal{D})$, denoted as $\mathcal{I} \models (\mathcal{P}, \mathcal{D})$, if $\mathcal{I}$ is a model of the KB $(\mathcal{O}, \mathcal{M}(\mathcal{D}))$.

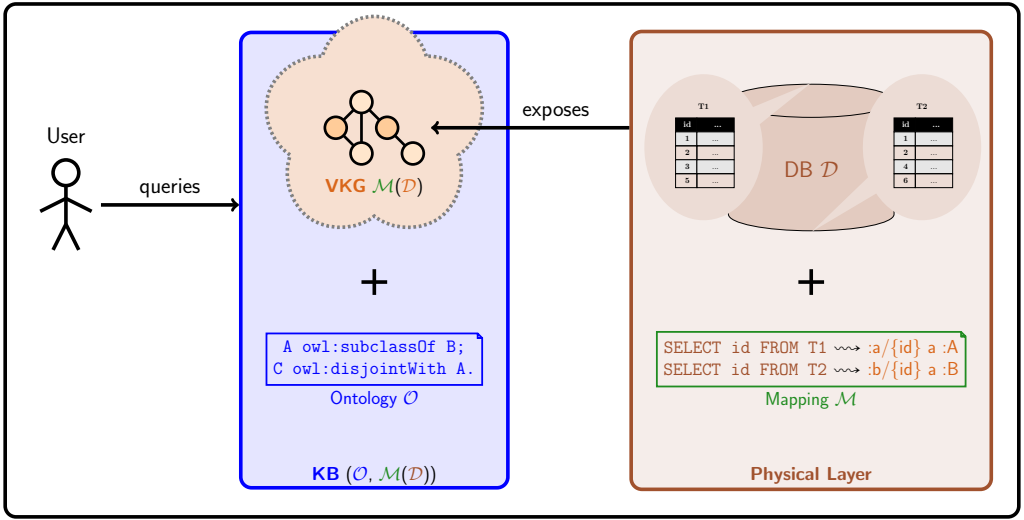
Note:

- In general, $(\mathcal{P}, \mathcal{D})$ has infinitely many models, and some of these might be infinite.
- However, for query answering, we do not need to compute such models.

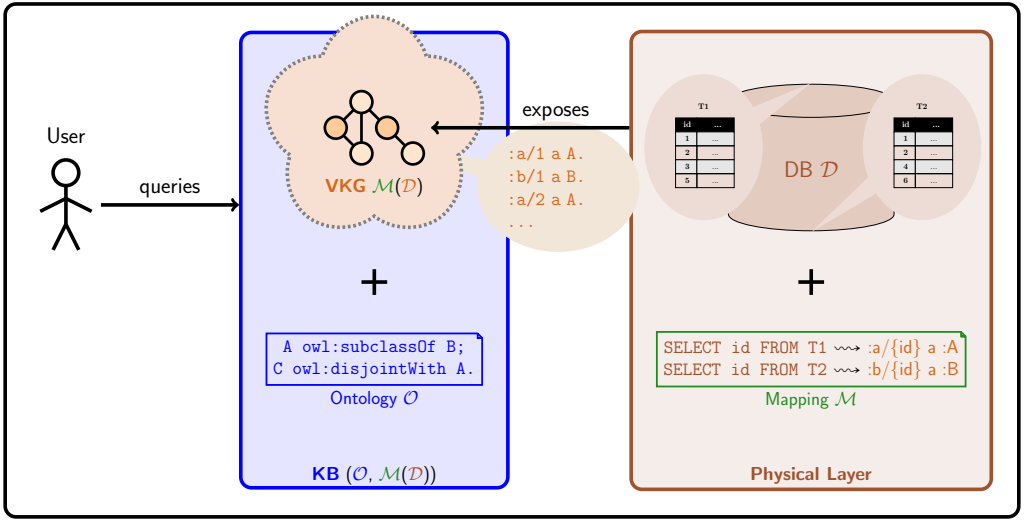
Example of VKG instance and corresponding KB



Example of VKG instance and corresponding KB



Example of VKG instance and corresponding KB



Query answering in KBs – Certain answers

In VKGs, we want to answer queries formulated over the ontology, by using the data provided by the data sources through the mapping.

Consider our formalization of VKGs and a VKG instance $\mathcal{J}$.

Certain answers cert($q, \mathcal{J}$) – Intuition

Given a VKG instance $\mathcal{J}$ and a query q over $\mathcal{J}$, the certain answers cert($q, \mathcal{J}$) to q over $\mathcal{J}$ are those answers to q that hold in every model of $\mathcal{J}$.

Certain answers cert($q, \mathcal{J}$) – Formal definition

Given a VKG instance $\mathcal{J} = (\mathcal{P}, \mathcal{D})$ and a query q over $\mathcal{J}$, a tuple $\vec{c}$ of constants in $\mathcal{M}(\mathcal{D})$ is a **certain answer** to q over $\mathcal{J}$, i.e., $\vec{c} \in \text{cert}(q, \mathcal{J})$, if for every model $\mathcal{I}$ of $\mathcal{J}$ we have that $\vec{c} \in \text{eval}(q, \mathcal{I})$.

Note: Each certain answer $\vec{c}$ is a tuple of constants in $\mathcal{M}(\mathcal{D})$, but when we evaluate q over an interpretation $\mathcal{I}$, it returns tuples of elements of $\Delta^{\mathcal{I}}$. Therefore, we should actually require that $\vec{c}^{\mathcal{I}} \in \text{eval}(q, \mathcal{I})$, and not that $\vec{c} \in \text{eval}(q, \mathcal{I})$.
However, due to the standard names assumption, we have that $\vec{c}^{\mathcal{I}} = \vec{c}$, so the two conditions are equivalent.

Query answering in KBs – Certain answers

In VKGs, we want to answer queries formulated over the ontology, by using the data provided by the data sources through the mapping.

Consider our formalization of VKGs and a VKG instance $\mathcal{J}$.

Certain answers cert($q, \mathcal{J}$) – Intuition

Given a VKG instance $\mathcal{J}$ and a query q over $\mathcal{J}$, the certain answers cert($q, \mathcal{J}$) to q over $\mathcal{J}$ are those answers to q that hold in every model of $\mathcal{J}$.

Certain answers cert($q, \mathcal{J}$) – Formal definition

Given a VKG instance $\mathcal{J} = (\mathcal{P}, \mathcal{D})$ and a query q over $\mathcal{J}$, a tuple $\vec{c}$ of constants in $\mathcal{M}(\mathcal{D})$ is a **certain answer** to q over $\mathcal{J}$, i.e., $\vec{c} \in \text{cert}(q, \mathcal{J})$, if for every model $\mathcal{I}$ of $\mathcal{J}$ we have that $\vec{c} \in \text{eval}(q, \mathcal{I})$.

Note: Each certain answer $\vec{c}$ is a tuple of constants in $\mathcal{M}(\mathcal{D})$, but when we evaluate q over an interpretation $\mathcal{I}$, it returns tuples of elements of $\Delta^{\mathcal{I}}$. Therefore, we should actually require that $\vec{c}^{\mathcal{I}} \in \text{eval}(q, \mathcal{I})$, and not that $\vec{c} \in \text{eval}(q, \mathcal{I})$.
However, due to the standard names assumption, we have that $\vec{c}^{\mathcal{I}} = \vec{c}$, so the two conditions are equivalent.

Query answering in KBs – Certain answers

In VKGs, we want to answer queries formulated over the ontology, by using the data provided by the data sources through the mapping.

Consider our formalization of VKGs and a VKG instance $\mathcal{J}$.

Certain answers cert($q, \mathcal{J}$) – Intuition

Given a VKG instance $\mathcal{J}$ and a query q over $\mathcal{J}$, the certain answers cert($q, \mathcal{J}$) to q over $\mathcal{J}$ are those answers to q that hold in every model of $\mathcal{J}$.

Certain answers cert($q, \mathcal{J}$) – Formal definition

Given a VKG instance $\mathcal{J} = (\mathcal{P}, \mathcal{D})$ and a query q over $\mathcal{J}$, a tuple $\vec{c}$ of constants in $\mathcal{M}(\mathcal{D})$ is a **certain answer** to q over $\mathcal{J}$, i.e., $\vec{c} \in \text{cert}(q, \mathcal{J})$, if for every model $\mathcal{I}$ of $\mathcal{J}$ we have that $\vec{c} \in \text{eval}(q, \mathcal{I})$.

Note: Each certain answer $\vec{c}$ is a tuple of constants in $\mathcal{M}(\mathcal{D})$, but when we evaluate q over an interpretation $\mathcal{I}$, it returns tuples of elements of $\Delta^{\mathcal{I}}$. Therefore, we should actually require that $\vec{c}^{\mathcal{I}} \in \text{eval}(q, \mathcal{I})$, and not that $\vec{c} \in \text{eval}(q, \mathcal{I})$.
However, due to the standard names assumption, we have that $\vec{c}^{\mathcal{I}} = \vec{c}$, so the two conditions are equivalent.

Query answering in KBs – Certain answers

In VKGs, we want to answer queries formulated over the ontology, by using the data provided by the data sources through the mapping.

Consider our formalization of VKGs and a VKG instance $\mathcal{J}$.

Certain answers cert($q, \mathcal{J}$) – Intuition

Given a VKG instance $\mathcal{J}$ and a query q over $\mathcal{J}$, the certain answers cert($q, \mathcal{J}$) to q over $\mathcal{J}$ are those answers to q that hold in every model of $\mathcal{J}$.

Certain answers cert($q, \mathcal{J}$) – Formal definition

Given a VKG instance $\mathcal{J} = (\mathcal{P}, \mathcal{D})$ and a query q over $\mathcal{J}$, a tuple $\vec{c}$ of constants in $\mathcal{M}(\mathcal{D})$ is a **certain answer** to q over $\mathcal{J}$, i.e., $\vec{c} \in \text{cert}(q, \mathcal{J})$, if for every model $\mathcal{I}$ of $\mathcal{J}$ we have that $\vec{c} \in \text{eval}(q, \mathcal{I})$.

Note: Each certain answer $\vec{c}$ is a tuple of constants in $\mathcal{M}(\mathcal{D})$, but when we evaluate q over an interpretation $\mathcal{I}$, it returns tuples of elements of $\Delta^{\mathcal{I}}$. Therefore, we should actually require that $\vec{c}^{\mathcal{I}} \in \text{eval}(q, \mathcal{I})$, and not that $\vec{c} \in \text{eval}(q, \mathcal{I})$.
However, due to the standard names assumption, we have that $\vec{c}^{\mathcal{I}} = \vec{c}$, so the two conditions are equivalent.

First-order rewritability

To make computing certain answers viable in practice, the VKG setting relies on reducing it to evaluating SQL (i.e., first-order logic) queries over the data.

Consider a VKG specification $\mathcal{P} = (\mathcal{O}, \mathcal{M}, \mathcal{S})$.

First-order rewritability

A query $r(\vec{x})$ is a **first-order rewriting** of a query $q(\vec{x})$ with respect to $\mathcal{P}$ if, for every source DB $\mathcal{D}$, certain answers to $q(\vec{x})$ over $(\mathcal{P}, \mathcal{D})$ = answers to $r(\vec{x})$ over $\mathcal{M}(\mathcal{D})$.

For OWL 2 QL ontologies and (a subset of) R2RML mappings, (core) SPARQL queries are first-order rewritable.

In other words, **in VKGs, we can compute the certain answers to a SPARQL query by evaluating over the sources its rewriting, which is a SQL query.**

First-order rewritability

To make computing certain answers viable in practice, the VKG setting relies on reducing it to evaluating SQL (i.e., first-order logic) queries over the data.

Consider a VKG specification $\mathcal{P} = (\mathcal{O}, \mathcal{M}, \mathcal{S})$.

First-order rewritability

A query $r(\vec{x})$ is a **first-order rewriting** of a query $q(\vec{x})$ with respect to $\mathcal{P}$ if, for every source DB $\mathcal{D}$,

certain answers to $q(\vec{x})$ over $(\mathcal{P}, \mathcal{D})$ = answers to $r(\vec{x})$ over $\mathcal{M}(\mathcal{D})$.

For OWL 2 QL ontologies and (a subset of) R2RML mappings,
(core) SPARQL queries are first-order rewritable.

In other words, **in VKGs, we can compute the certain answers to a SPARQL query by evaluating over the sources its rewriting, which is a SQL query.**

First-order rewritability

To make computing certain answers viable in practice, the VKG setting relies on reducing it to evaluating SQL (i.e., first-order logic) queries over the data.

Consider a VKG specification $\mathcal{P} = (\mathcal{O}, \mathcal{M}, \mathcal{S})$.

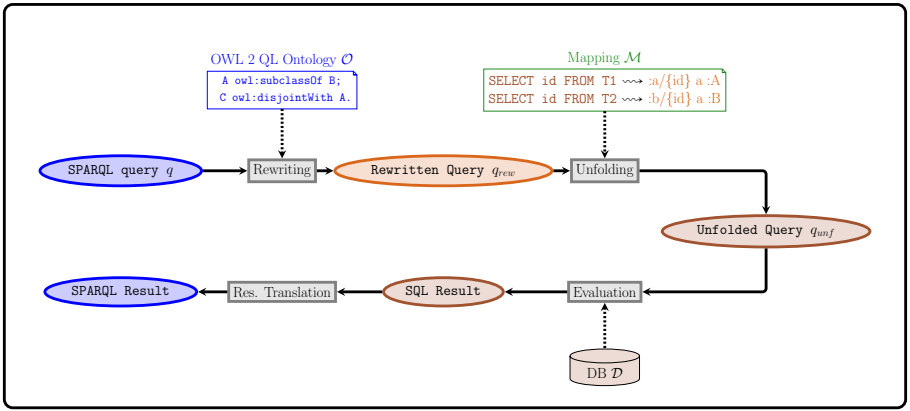
First-order rewritability

A query $r(\vec{x})$ is a **first-order rewriting** of a query $q(\vec{x})$ with respect to $\mathcal{P}$ if, for every source DB $\mathcal{D}$,
certain answers to $q(\vec{x})$ over $(\mathcal{P}, \mathcal{D})$ = answers to $r(\vec{x})$ over $\mathcal{M}(\mathcal{D})$.

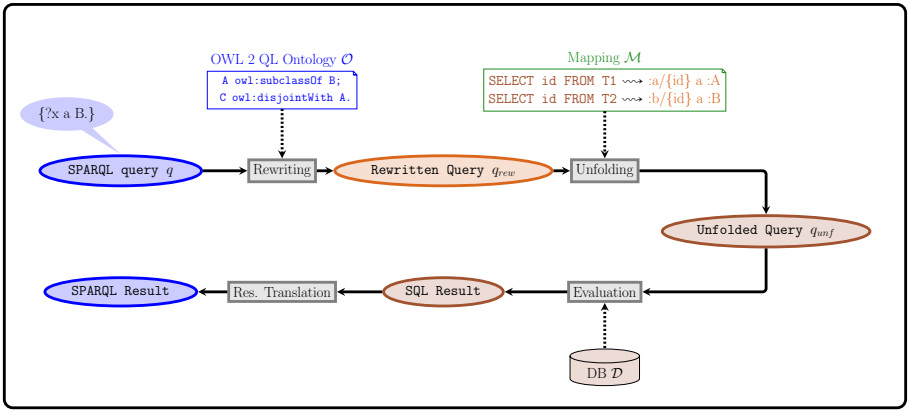
For OWL 2 QL ontologies and (a subset of) R2RML mappings,
(core) SPARQL queries are first-order rewritable.

In other words, **in VKGs, we can compute the certain answers to a SPARQL query by evaluating over the sources its rewriting, which is a SQL query.**

Under the hood: Query evaluation process I

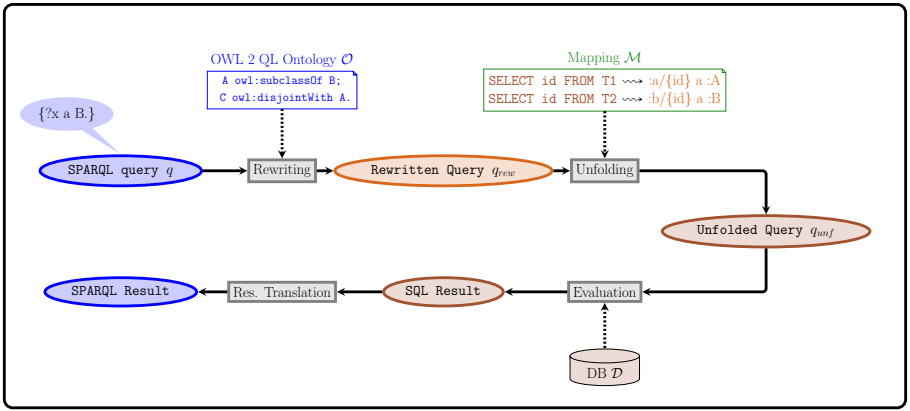


Under the hood: Query evaluation process II



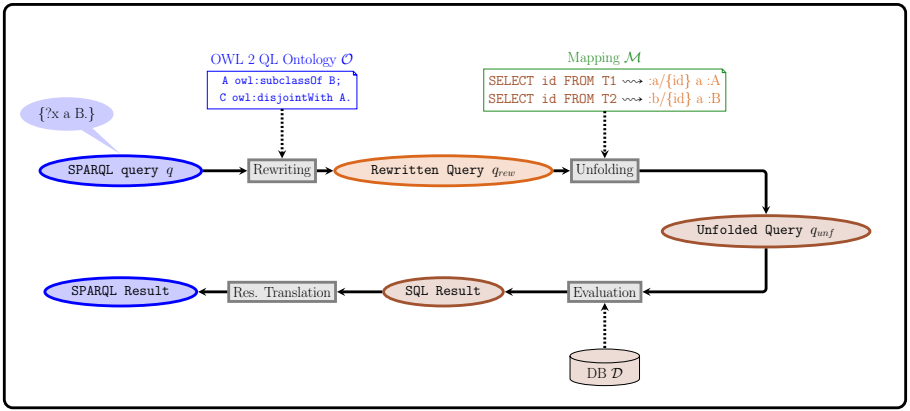
- **Problem:** Find $Ans := cert(q, (\mathcal{P}, \mathcal{D})) := cert(q, (\mathcal{O}, \mathcal{M}(\mathcal{D})))$ **(certain answers)**
  - with cert(...) defined as $\bigcap_{\mathcal{I} \models (\mathcal{O}, \mathcal{M}(\mathcal{D}))} \text{eval}(q, \mathcal{I})$ (query evaluation)

Under the hood: Query evaluation process II



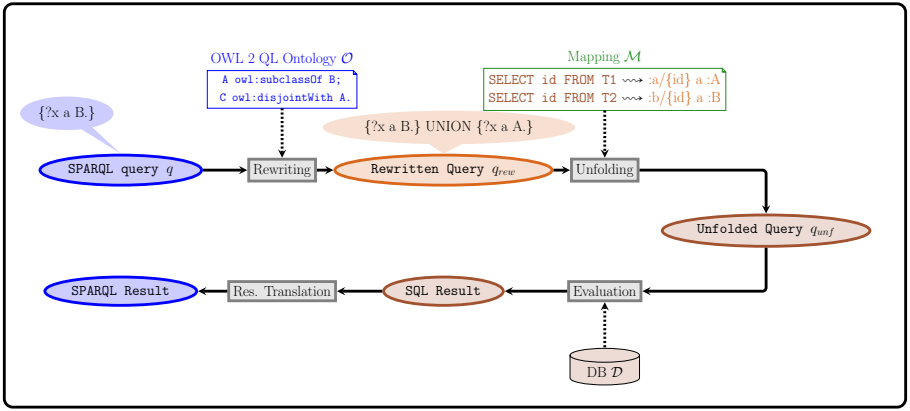
- **Problem:** Find $Ans := \text{cert}(q, (\mathcal{P}, \mathcal{D})) := \text{cert}(q, (\mathcal{O}, \mathcal{M}(\mathcal{D})))$ **(certain answers)**
  - with cert(...) defined as $\bigcap_{\mathcal{I} \models (\mathcal{O}, \mathcal{M}(\mathcal{D}))} \text{eval}(q, \mathcal{I})$ (query evaluation)

Under the hood: Query evaluation process II



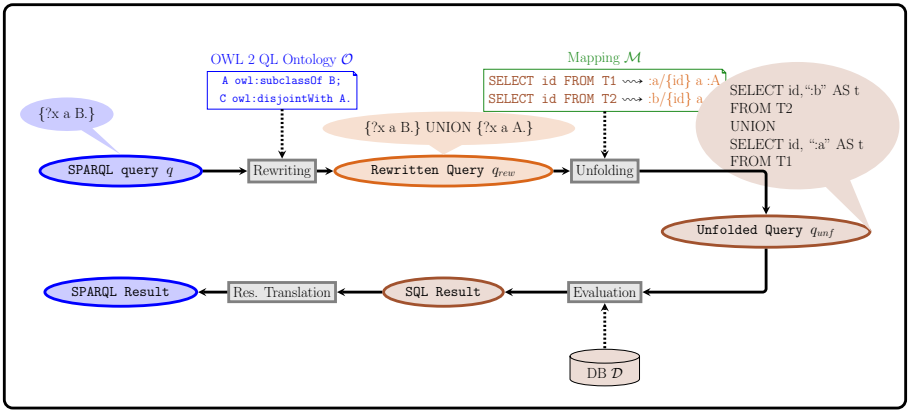
- **Problem:** Find $Ans := \text{cert}(q, (\mathcal{P}, \mathcal{D})) := \text{cert}(q, (\mathcal{O}, \mathcal{M}(\mathcal{D})))$ **(certain answers)**
  - with cert(...) defined as $\bigcap_{\mathcal{I} \models (\mathcal{O}, \mathcal{M}(\mathcal{D}))} \text{eval}(q, \mathcal{I})$ **(query evaluation)**

Under the hood: Query evaluation process III



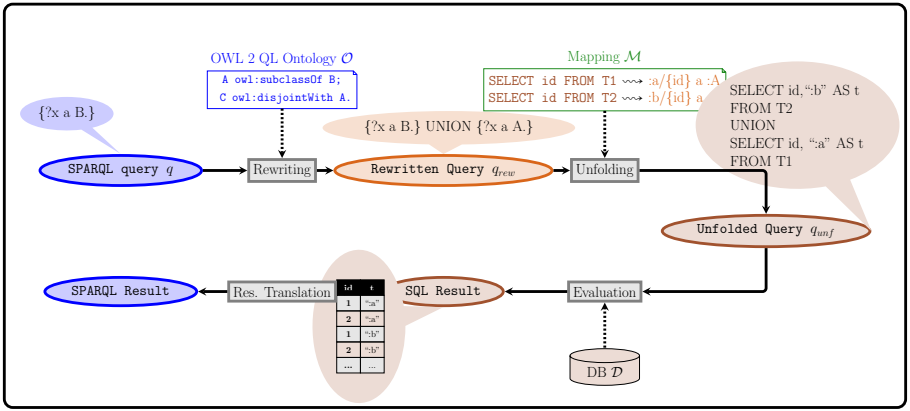
- **Problem:** Find $Ans := \text{cert}(q, (\mathcal{P}, \mathcal{D}))$
- q_{rew} is a **rewriting** $\rightsquigarrow Ans = \text{eval}(q_{rew}, can(\mathcal{M}(\mathcal{D})))$ [C., De Giacomo, et al. 2007]
  where $can(\mathcal{M}(\mathcal{D}))$ denotes the (unique) model for the VKG

Under the hood: Query evaluation process IV



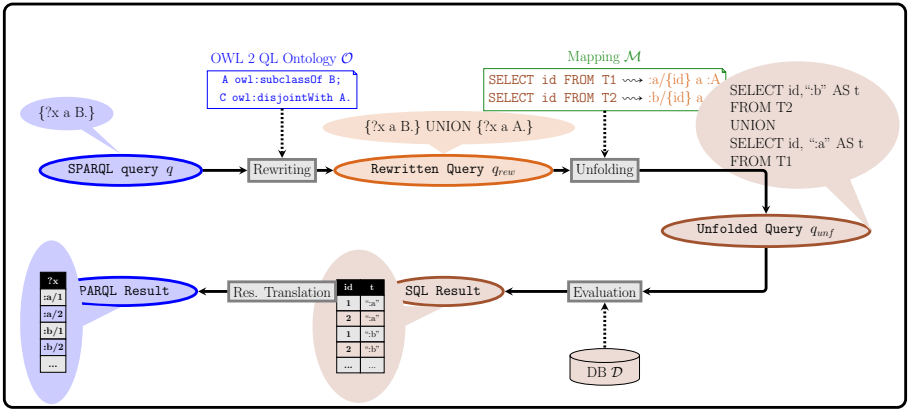
- **Problem:** Find $Ans := \text{cert}(q, (\mathcal{P}, \mathcal{D}))$
- q_{rew} is a **rewriting** $\rightsquigarrow Ans = \text{eval}(q_{rew}, can(\mathcal{M}(\mathcal{D})))$ [C., De Giacomo, et al. 2007]
- q_{unf} is a **translation** $\rightsquigarrow Ans = \text{eval}(q_{unf}, \mathcal{D})$ [Poggi et al. 2008]

Under the hood: Query evaluation process V



- **Problem:** Find $Ans := \text{cert}(q, (\mathcal{P}, \mathcal{D}))$
- q_{rew} is a **rewriting** $\rightsquigarrow Ans = \text{eval}(q_{rew}, can(\mathcal{M}(\mathcal{D})))$ [C., De Giacomo, et al. 2007]
- q_{unf} is a **translation** $\rightsquigarrow Ans = \text{eval}(q_{unf}, \mathcal{D})$ [Poggi et al. 2008]

Under the hood: Query evaluation process VI



- **Problem:** Find $Ans := \text{cert}(q, (\mathcal{P}, \mathcal{D}))$
- q_{rew} is a **rewriting** $\rightsquigarrow Ans = \text{eval}(q_{rew}, can(\mathcal{M}(\mathcal{D})))$ [C., De Giacomo, et al. 2007]
- q_{unf} is a **translation** $\rightsquigarrow Ans = \text{eval}(q_{unf}, \mathcal{D})$ [Poggi et al. 2008]

Outline

Outline

Who provides the ontology?

- Designing an ontology is not an easy task.
- In many domains (e.g., the biomedical one) ontologies are developed independently by trained experts and are already available to be re-used.
- Having “standardized ontologies” enables interoperability across different data sources.
- However, ontology design is a well investigated task, and methodologies and supporting tools are readily available. See, e.g.,
  - the series of *Workshops on Ontology Design Patterns* `http://ontologydesignpatterns.org/`;
  - the OntoClean methodology for ontology analysis based on formal, domain-independent properties of classes [Guarino & Welty 2009].

Who provides the mappings?

VKG mappings:

- Map complex queries to complex queries – cf. GLAV relational mappings [Lenzerini 2002].
- Overcome the abstraction mismatch between relational data and target ontology.
- Are inherently more sophisticated than mappings for schema matching [Rahm & Bernstein 2001] and ontology matching [Euzenat & Shvaiko 2007].

As a consequence:

- Management of VKG mappings is an essentially manual effort that is **labor-intensive** and **error-prone**.
- Requires highly-skilled professionals [Spanos et al. 2012].
- Writing mappings is challenging in terms of semantics, correctness, and performance.

Designing and managing mappings is the most critical bottleneck for the adoption of the VKG approach.

Who provides the mappings?

VKG mappings:

- Map complex queries to complex queries – cf. GLAV relational mappings [Lenzerini 2002].
- Overcome the abstraction mismatch between relational data and target ontology.
- Are inherently more sophisticated than mappings for schema matching [Rahm & Bernstein 2001] and ontology matching [Euzenat & Shvaiko 2007].

As a consequence:

- Management of VKG mappings is an essentially manual effort that is **labor-intensive** and **error-prone**.
- Requires highly-skilled professionals [Spanos et al. 2012].
- Writing mappings is challenging in terms of semantics, correctness, and performance.

Designing and managing mappings is the most critical bottleneck for the adoption of the VKG approach.

Who provides the mappings?

VKG mappings:

- Map complex queries to complex queries – cf. GLAV relational mappings [Lenzerini 2002].
- Overcome the abstraction mismatch between relational data and target ontology.
- Are inherently more sophisticated than mappings for schema matching [Rahm & Bernstein 2001] and ontology matching [Euzenat & Shvaiko 2007].

As a consequence:

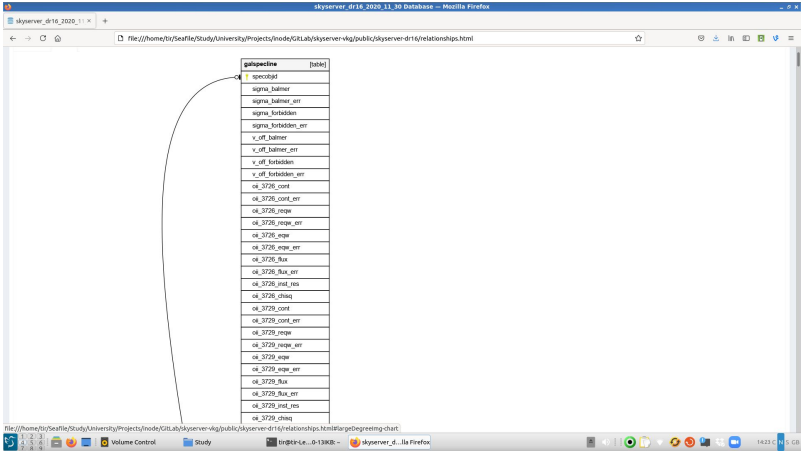
- Management of VKG mappings is an essentially manual effort that is **labor-intensive** and **error-prone**.
- Requires highly-skilled professionals [Spanos et al. 2012].
- Writing mappings is challenging in terms of semantics, correctness, and performance.

Designing and managing mappings is the most critical bottleneck
for the adoption of the VKG approach.

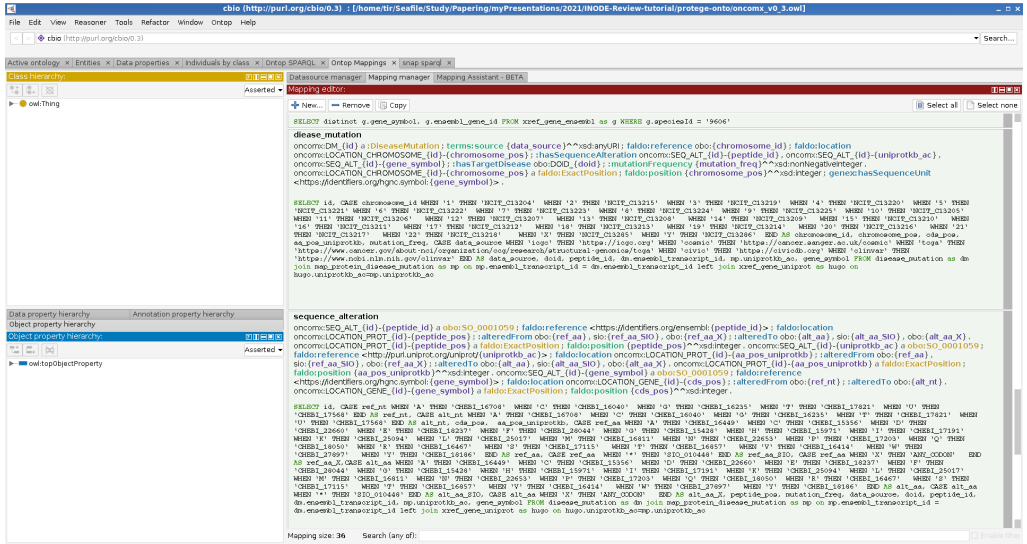
Who provides the mapping?

Writing mappings manually is a **time-consuming** and **error-prone** task.

Who provides the mapping?



Who provides the mapping?



Designing VKG mappings

A good design of mappings in VKGs is critical in ensuring that:

- the resulting VKG specification captures correctly the domain semantics, and
- queries posed over a VKG instance can be answered efficiently.

In designing the mapping assertions, we should take into account the following:

- For each atom in the target part, the source query should be the **simplest SQL query** that retrieves the data that is necessary to populate that atom.
- In particular, we should **avoid unnecessary joins** in the source query.
- We should combine two (or more) atoms in a single mapping assertion only if they require the same source query.
- We need to pay attention to the form of the IRI-templates, to ensure that the "same" ontology object retrieved through multiple mappings is constructed with the same IRI-template.

However, these observations in general are not sufficient to ensure a good mapping design.

Designing VKG mappings

A good design of mappings in VKGs is critical in ensuring that:

- the resulting VKG specification captures correctly the domain semantics, and
- queries posed over a VKG instance can be answered efficiently.

In designing the mapping assertions, we should take into account the following:

- For each atom in the target part, the source query should be the **simplest SQL query** that retrieves the data that is necessary to populate that atom.
- In particular, we should **avoid unnecessary joins** in the source query.
- We should combine two (or more) atoms in a single mapping assertion only if they require the same source query.
- We need to pay attention to the form of the IRI-templates, to ensure that the “same” ontology object retrieved through multiple mappings is constructed with the same IRI-template.

However, these observations in general are not sufficient to ensure a good mapping design.

Designing VKG mappings

A good design of mappings in VKGs is critical in ensuring that:

- the resulting VKG specification captures correctly the domain semantics, and
- queries posed over a VKG instance can be answered efficiently.

In designing the mapping assertions, we should take into account the following:

- For each atom in the target part, the source query should be the **simplest SQL query** that retrieves the data that is necessary to populate that atom.
- In particular, we should **avoid unnecessary joins** in the source query.
- We should combine two (or more) atoms in a single mapping assertion only if they require the same source query.
- We need to pay attention to the form of the IRI-templates, to ensure that the “same” ontology object retrieved through multiple mappings is constructed with the same IRI-template.

However, these observations in general are not sufficient to ensure a good mapping design.

Patterns in data sources

- In order to simplify the task of mapping design, it is convenient to identify whether the data source satisfies certain common patterns.
- Each such data pattern can be captured in a sort of “standard” way through a specific form of mapping assertions, combined with some specific form of ontology axiom.
- The presence of a pattern in a data source, and hence the applicability of the corresponding standard encoding into mappings (and ontology axioms), is signaled by the presence of some (combination of) **constraints holding over the relational tables**.
- Notice that such constraints might hold:
  - either because they are explicitly declared in the database, and hence enforced by the DBMS,
  - or because they are implied by the semantics of the domain, even though they might not be declared explicitly in the database.

Looking at database design principles

In relational database design, **well-established conceptual modeling principles** and **methodologies** are usually employed.

- The resulting schema should suitably reflects the application domain at hand.
- This design phase relies on semantically-rich representations such as ER diagrams.
- However, these representations, typically:
  - get lost during deployment, since they are not conveyed together with the database itself, or
  - quickly get outdated due to continuous adjustments triggered by changing requirements.

Key Observation

While the relational model may be semantically-poor with respect to ontological models, the original semantically-rich design of the application domain **leaves recognizable footprints** that can be converted into ontological mapping patterns.

Looking at database design principles

In relational database design, **well-established conceptual modeling principles** and **methodologies** are usually employed.

- The resulting schema should suitably reflects the application domain at hand.
- This design phase relies on semantically-rich representations such as ER diagrams.
- However, these representations, typically:
  - get lost during deployment, since they are not conveyed together with the database itself, or
  - quickly get outdated due to continuous adjustments triggered by changing requirements.

Key Observation

While the relational model may be semantically-poor with respect to ontological models, the original semantically-rich design of the application domain **leaves recognizable footprints** that can be converted into ontological mapping patterns.

Looking at database design principles

In relational database design, **well-established conceptual modeling principles** and **methodologies** are usually employed.

- The resulting schema should suitably reflects the application domain at hand.
- This design phase relies on semantically-rich representations such as ER diagrams.
- However, these representations, typically:
  - get lost during deployment, since they are not conveyed together with the database itself, or
  - quickly get outdated due to continuous adjustments triggered by changing requirements.

Key Observation

While the relational model may be semantically-poor with respect to ontological models, the original semantically-rich design of the application domain **leaves recognizable footprints** that can be converted into ontological mapping patterns.

Outline

Elements characterizing a VKG mapping pattern

Therefore, in designing VKG mapping patterns, we draw an explicit and precise connection with conceptual modeling practices found in DB design, while exploiting all of:

- the relational schema with its constraints
- the conceptual schema at the basis of the relational schema
- extensional data stored in the DB (when available)
- the domain knowledge that is encoded in ontology axioms

Catalog of mapping patterns

To come up with a catalog of mapping patterns, we can rely on well-established methodologies and patterns studied in:

- data management – e.g., W3C Direct Mapping Specification [Arenas et al. 2012] and extensions
- data analysis – e.g., algorithms for discovering dependencies, and
- conceptual modeling

The specification of each pattern includes:

- the three components of a VKG specification: DB schema, ontology, mapping between the two;
- the conceptual schema of the domain of interest;
- underlying data, when available.

Note that the patterns do not fix what is given as input and what is produced as output, but simply describe how the different elements relate to each other.

Catalog of mapping patterns

To come up with a catalog of mapping patterns, we can rely on well-established methodologies and patterns studied in:

- data management – e.g., W3C Direct Mapping Specification [Arenas et al. 2012] and extensions
- data analysis – e.g., algorithms for discovering dependencies, and
- conceptual modeling

The specification of each pattern includes:

- the three components of a VKG specification: DB schema, ontology, mapping between the two;
- the conceptual schema of the domain of interest;
- underlying data, when available.

Note that the patterns do not fix what is given as input and what is produced as output, but simply describe how the different elements relate to each other.

Catalog of mapping patterns

To come up with a catalog of mapping patterns, we can rely on well-established methodologies and patterns studied in:

- data management – e.g., W3C Direct Mapping Specification [Arenas et al. 2012] and extensions
- data analysis – e.g., algorithms for discovering dependencies, and
- conceptual modeling

The specification of each pattern includes:

- the three components of a VKG specification: DB schema, ontology, mapping between the two;
- the conceptual schema of the domain of interest;
- underlying data, when available.

Note that the patterns do not fix what is given as input and what is produced as output, but simply describe how the different elements relate to each other.

Two major groups of mapping patterns

Schema-driven patterns

Are shaped by the structure of the DB schema and its explicit constraints.

Data-driven patterns

- Consider also constraints emerging from specific configurations of the data in the DB.
- For each schema-driven pattern, we identify a data-driven version:
  The constraints over the schema are not explicitly specified, but hold in the data.
- We provide also data-driven patterns that do not have a schema-driven counterpart.

Some patterns come with **views over the DB-schema**:

- Views reveal structures over the DB-schema, when the pattern is applied.
- Views can be used to identify the applicability of further patterns.

Two major groups of mapping patterns

Schema-driven patterns

Are shaped by the structure of the DB schema and its explicit constraints.

Data-driven patterns

- Consider also constraints emerging from specific configurations of the data in the DB.
- For each schema-driven pattern, we identify a data-driven version:
  The constraints over the schema are not explicitly specified, but hold in the data.
- We provide also data-driven patterns that do not have a schema-driven counterpart.

Some patterns come with **views over the DB-schema**:

- Views reveal structures over the DB-schema, when the pattern is applied.
- Views can be used to identify the applicability of further patterns.

Two major groups of mapping patterns

Schema-driven patterns

Are shaped by the structure of the DB schema and its explicit constraints.

Data-driven patterns

- Consider also constraints emerging from specific configurations of the data in the DB.
- For each schema-driven pattern, we identify a data-driven version:
  The constraints over the schema are not explicitly specified, but hold in the data.
- We provide also data-driven patterns that do not have a schema-driven counterpart.

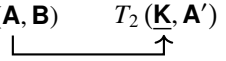
Some patterns come with **views over the DB-schema**:

- Views reveal structures over the DB-schema, when the pattern is applied.
- Views can be used to identify the applicability of further patterns.

Constraints on the data

When defining the mapping patterns, we consider the following types of constraints:

- **Primary key constraint**: denoted $T(\underline{\mathbf{K}}, \mathbf{A})$, where **K** is a set of attributes that form the primary key of relation T, and **A** are the remaining attributes of T.
- **Key constraint**: denoted $\mathit{unique}_T(\mathbf{K})$, where **K** is a set of attributes that form a key of the relation T on which the attributes **K** are defined.
- **Foreign key constraint**: denoted $T_1[\mathbf{A}] \subseteq T_2[\mathbf{K}]$, where **A** is a set of attributes of relation T_1 and **K** is a key (typically, the primary key) of relation T_2. For convenience, we represent the constraint $T_1[\mathbf{A}] \subseteq T_2[\mathbf{K}]$ by drawing an arrow from **A** in the schema of T_1 to **K** in the schema of T_2, i.e.,

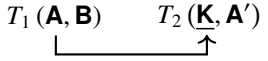


Note: We denote single attributes of a relational table using normal math font (e.g., A), while we use boldface to indicate sets of attributes (e.g., **A** or **K**).

Constraints on the data

When defining the mapping patterns, we consider the following types of constraints:

- **Primary key constraint**: denoted $T(\underline{\mathbf{K}}, \mathbf{A})$, where **K** is a set of attributes that form the primary key of relation T, and **A** are the remaining attributes of T.
- **Key constraint**: denoted $\mathit{unique}_T(\mathbf{K})$, where **K** is a set of attributes that form a key of the relation T on which the attributes **K** are defined.
- **Foreign key constraint**: denoted $T_1[\mathbf{A}] \subseteq T_2[\mathbf{K}]$, where **A** is a set of attributes of relation T_1 and **K** is a key (typically, the primary key) of relation T_2. For convenience, we represent the constraint $T_1[\mathbf{A}] \subseteq T_2[\mathbf{K}]$ by drawing an arrow from **A** in the schema of T_1 to **K** in the schema of T_2, i.e.,



Note: We denote single attributes of a relational table using normal math font (e.g., A), while we use boldface to indicate sets of attributes (e.g., **A** or **K**).

Types of mapping patterns

We have defined several types of mapping patterns.

- Entity (MpE)
- Relationship (MpR)
- Relationship with Identifier Alignment (MpRa)
- Relationship with Merging (MpRm)
- 1-1 Relationship with Merging (MpR11m)
- Entity with Weak Identification (MpEw)
- Reified Relationship (MpRR)
- Hierarchy (MpH)
- Hierarchy with Identifier Alignment (MpHa)
- Clustering Entity to Class (MpCE2C) / Data Property / Object Property

We present each mapping pattern by specifying the following four components:

1. The constraints over the relational schema/data that make the patterns applicable.
2. A possible conceptual schema (specified as an Entity-Relationship diagram) that corresponds to such constraints.

   The elements that are directly affected by the pattern and that give rise to the mapping assertions are outlined in red.
3. The resulting mapping assertion(s) (given as source and target parts).
4. The ontology axioms that should hold.

Note: In the following, we make use of IRI-templates of the form "`:E/{K}`", where we assume that "`:E/`" is a prefix that is specific for the instances of a class C_E.

Types of mapping patterns

We have defined several types of mapping patterns. We discuss here **some significant ones**:

- Entity (MpE)
- Relationship (MpR)
- Relationship with Identifier Alignment (MpRa)
- Relationship with Merging (MpRm)
- 1-1 Relationship with Merging (MpR11m)
- Entity with Weak Identification (MpEw)
- Reified Relationship (MpRR)
- Hierarchy (MpH)
- Hierarchy with Identifier Alignment (MpHa)
- Clustering Entity to Class (MpCE2C) / Data Property / Object Property

We present each mapping pattern by specifying the following four components:

1. The constraints over the relational schema/data that make the patterns applicable.
2. A possible conceptual schema (specified as an Entity-Relationship diagram) that corresponds to such constraints.

   The elements that are directly affected by the pattern and that give rise to the mapping assertions are outlined in red.
3. The resulting mapping assertion(s) (given as source and target parts).
4. The ontology axioms that should hold.

Note: In the following, we make use of IRI-templates of the form "`:E/{K}`", where we assume that "`:E/`" is a prefix that is specific for the instances of a class C_E.

Types of mapping patterns

We have defined several types of mapping patterns.

- Entity (MpE)
- Relationship (MpR)
- Relationship with Identifier Alignment (MpRa)
- Relationship with Merging (MpRm)
- 1-1 Relationship with Merging (MpR11m)
- Entity with Weak Identification (MpEw)
- Reified Relationship (MpRR)
- Hierarchy (MpH)
- Hierarchy with Identifier Alignment (MpHa)
- Clustering Entity to Class (MpCE2C) / Data Property / Object Property

We present each mapping pattern by specifying the following four components:

1. The constraints over the relational schema/data that make the patterns applicable.
2. A possible conceptual schema (specified as an Entity-Relationship diagram) that corresponds to such constraints.

   The elements that are directly affected by the pattern and that give rise to the mapping assertions are outlined in red.
3. The resulting mapping assertion(s) (given as source and target parts).
4. The ontology axioms that should hold.

Note: In the following, we make use of IRI-templates of the form "`:E/{K}`", where we assume that "`:E/`" is a prefix that is specific for the instances of a class C_E.

Types of mapping patterns

We have defined several types of mapping patterns.

- Entity (MpE)
- Relationship (MpR)
- Relationship with Identifier Alignment (MpRa)
- Relationship with Merging (MpRm)
- 1-1 Relationship with Merging (MpR11m)
- Entity with Weak Identification (MpEw)
- Reified Relationship (MpRR)
- Hierarchy (MpH)
- Hierarchy with Identifier Alignment (MpHa)
- Clustering Entity to Class (MpCE2C) / Data Property / Object Property

We present each mapping pattern by specifying the following four components:

1. The constraints over the relational schema/data that make the patterns applicable.
2. A possible conceptual schema (specified as an Entity-Relationship diagram) that corresponds to such constraints.

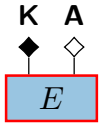
   The elements that are directly affected by the pattern and that give rise to the mapping assertions are outlined in red.
3. The resulting mapping assertion(s) (given as source and target parts).
4. The ontology axioms that should hold.

Note: In the following, we make use of IRI-templates of the form "`:E/{K}`", where we assume that "`:E/`" is a prefix that is specific for the instances of a class C_E.

Mapping pattern: Entity (MpE)

Relational schema and constraints:

$T_E(\underline{\mathbf{K}}, \mathbf{A})$



Mapping assertion:

$s : T_E$

$t :$ `:E/{K} rdf:type` C_E .
`{ :E/{K}` d_A `{A} . }`$_{A \in \mathbf{K} \cup \mathbf{A}}$

Ontology axioms:

$$\left\{ \begin{array}{l} \exists d_A \sqsubseteq C_E \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_E \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{K} \cup \mathbf{A}}$$

For each optional attribute A', add an $opt(A')$ indication to the relational schema and drop the corresponding axiom $C_E \sqsubseteq \exists d_{A'}$ from the ontology.

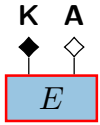
For the application of the mapping pattern, we observe the following:

- The pattern considers a single table T_E with primary key $\underline{\mathbf{K}}$ and other relevant attributes $\mathbf{A}$.
- The pattern captures how T_E is mapped into a corresponding class C_E.
- The primary key $\underline{\mathbf{K}}$ of T_E is used to construct the objects that are instances of C_E, using a template `:E/{K}` specific for C_E.
- Each relevant attribute of T_E is mapped to a data property of C_E.

Mapping pattern: Entity (MpE)

Relational schema and constraints:

$T_E(\underline{\mathbf{K}}, \mathbf{A})$



Mapping assertion:

$s : T_E$

t : `:E/{K} rdf:type` C_E .
`{ :E/{K}` d_A `{A} . }`$_{A \in \mathbf{K} \cup \mathbf{A}}$

Ontology axioms:

$$\left\{ \begin{array}{l} \exists d_A \sqsubseteq C_E \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_E \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{K} \cup \mathbf{A}}$$

For each optional attribute A', add an $opt(A')$ indication to the relational schema and drop the corresponding axiom $C_E \sqsubseteq \exists d_{A'}$ from the ontology.

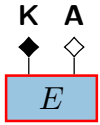
For the application of the mapping pattern, we observe the following:

- The pattern considers a single table T_E with primary key $\underline{\mathbf{K}}$ and other relevant attributes $\mathbf{A}$.
- The pattern captures how T_E is mapped into a corresponding class C_E.
- The primary key $\underline{\mathbf{K}}$ of T_E is used to construct the objects that are instances of C_E, using a template `:E/{K}` specific for C_E.
- Each relevant attribute of T_E is mapped to a data property of C_E.

Mapping pattern: Entity (MpE)

Relational schema and constraints:

$T_E(\underline{\mathbf{K}}, \mathbf{A})$



Mapping assertion:

$s : T_E$

t : :E/{**K**} rdf:type C_E .
{ :E/{**K**} d_A {A} . }$_{A \in \mathbf{K} \cup \mathbf{A}}$

Ontology axioms:

$$\left\{ \begin{array}{l} \exists d_A \sqsubseteq C_E \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_E \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{K} \cup \mathbf{A}}$$

For each optional attribute A', add an *opt*(A') indication to the relational schema and drop the corresponding axiom $C_E \sqsubseteq \exists d_{A'}$ from the ontology.

For the application of the mapping pattern, we observe the following:

- The pattern considers a single table T_E with primary key **K** and other relevant attributes **A**.
- The pattern captures how T_E is mapped into a corresponding class C_E.
- The primary key **K** of T_E is used to construct the objects that are instances of C_E, using a template :E/{**K**} specific for C_E.
- Each relevant attribute of T_E is mapped to a data property of C_E.

Mapping pattern: Entity (MpE) – Example

Consider a TClient table containing ssns of clients, together with name, dateOfBirth, and hobbies as additional attributes.

```
TClient(ssn,name,dateOfBirth,hobbies)
```

Mapping: TClient is mapped to a Client class using the attribute ssn to construct the IRIs for its instances.
In addition, the ssn, name, and dateOfBirth attributes are used to populate in the object position the three data properties ssn, name, and dob, respectively. The attribute hobbies is ignored.

```
mappingId MClient
source    SELECT ssn, name, dateOfBirth FROM TClient
target    :C/{ssn} rdf:type :Client ;
                   :ssn {ssn} ;
                   :name {name} ;
                   :dob {dateOfBirth} .
```

Mapping pattern: Entity (MpE) – Example

Consider a TClient table containing ssns of clients, together with name, dateOfBirth, and hobbies as additional attributes.

```
TClient(ssn,name,dateOfBirth,hobbies)
```

Mapping: TClient is mapped to a Client class using the attribute ssn to construct the IRIs for its instances.
In addition, the ssn, name, and dateOfBirth attributes are used to populate in the object position the three data properties ssn, name, and dob, respectively. The attribute hobbies is ignored.

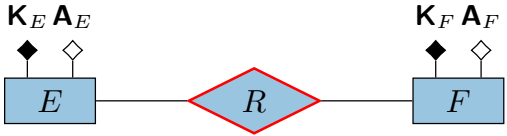
```
mappingId MClient
source    SELECT ssn, name, dateOfBirth FROM TClient
target    :C/{ssn} rdf:type :Client ;
                   :ssn {ssn} ;
                   :name {name} ;
                   :dob {dateOfBirth} .
```

Mapping pattern: Relationship (MpR) – Case of (0,n)–(0,n) cardinalities

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}})$



Mapping assertion:

$s : T_R$

$t :$:E/{$\mathbf{K}_{RE}$} p_R :F/{$\mathbf{K}_{RF}$} .

Ontology axioms:

$\exists p_R \sqsubseteq C_E$

$\exists p_R^- \sqsubseteq C_F$

For the application of the mapping pattern, we observe the following:

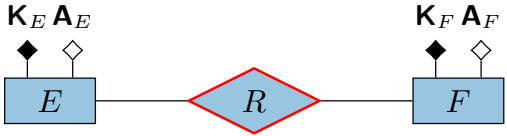
- This pattern considers three tables T_R, T_E, and T_F.
- The primary key of T_R is partitioned into two parts $\mathbf{K}_{RE}$ and $\mathbf{K}_{RF}$ that are foreign keys to T_E and T_F, respectively.
- T_R has no additional (relevant) attributes.
- The pattern captures how T_R is mapped to an object property p_R, using the two parts $\mathbf{K}_{RE}$ and $\mathbf{K}_{RF}$ of the primary key to construct respectively the subject and the object of the triples in p_R.

Mapping pattern: Relationship (MpR) – Case of (0,n)–(0,n) cardinalities

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}})$



Mapping assertion:

$s : T_R$

$t :$:E/{$\mathbf{K}_{RE}$} p_R :F/{$\mathbf{K}_{RF}$} .

Ontology axioms:

$\exists p_R \sqsubseteq C_E$

$\exists p_R^- \sqsubseteq C_F$

For the application of the mapping pattern, we observe the following:

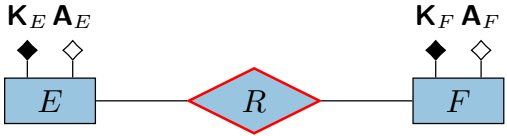
- This pattern considers three tables T_R, T_E, and T_F.
- The primary key of T_R is partitioned into two parts $\mathbf{K}_{RE}$ and $\mathbf{K}_{RF}$ that are foreign keys to T_E and T_F, respectively.
- T_R has no additional (relevant) attributes.
- The pattern captures how T_R is mapped to an object property p_R, using the two parts $\mathbf{K}_{RE}$ and $\mathbf{K}_{RF}$ of the primary key to construct respectively the subject and the object of the triples in p_R.

Mapping pattern: Relationship (MpR) – Case of (0,n)–(0,n) cardinalities

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}})$



Mapping assertion:

$s : T_R$

$t :$:E/{$\mathbf{K}_{RE}$} p_R :F/{$\mathbf{K}_{RF}$} .

Ontology axioms:

$\exists p_R \sqsubseteq C_E$

$\exists p_R^- \sqsubseteq C_F$

For the application of the mapping pattern, we observe the following:

- This pattern considers three tables T_R, T_E, and T_F.
- The primary key of T_R is partitioned into two parts $\mathbf{K}_{RE}$ and $\mathbf{K}_{RF}$ that are foreign keys to T_E and T_F, respectively.
- T_R has no additional (relevant) attributes.
- The pattern captures how T_R is mapped to an object property p_R, using the two parts $\mathbf{K}_{RE}$ and $\mathbf{K}_{RF}$ of the primary key to construct respectively the subject and the object of the triples in p_R.

Mapping pattern: Relationship (MpR) – Example

An additional TAddress table in the client registry stores the addresses at which each client can be reached, and such table has a foreign key to a table TLocation storing locations using attributes city and street.

```
TClient(ssn,name,dateOfBirth,hobbies)
TLocation(city,street)
TAddress(client,locCity,locStreet)
    FK: TAddress[client] -> Tclient[ssn]
    FK: TAddress[locCity,locStreet] -> TLocation[city,street]
```

Mapping: The TAddress table is mapped to an address object property, for which the ontology asserts that the domain is the class Client and the range an additional class Location, corresponding to the TLocation table.

```
mappingId MAddress
source    SELECT client, locCity, locStreet FROM TAddress
target    :C/{client} :address :L/{locCity}/{locStreet} .
```

Mapping pattern: Relationship (MpR) – Example

An additional TAddress table in the client registry stores the addresses at which each client can be reached, and such table has a foreign key to a table TLocation storing locations using attributes city and street.

```
TClient(ssn,name,dateOfBirth,hobbies)
TLocation(city,street)
TAddress(client,locCity,locStreet)
    FK: TAddress[client] -> Tclient[ssn]
    FK: TAddress[locCity,locStreet] -> TLocation[city,street]
```

Mapping: The TAddress table is mapped to an address object property, for which the ontology asserts that the domain is the class Client and the range an additional class Location, corresponding to the TLocation table.

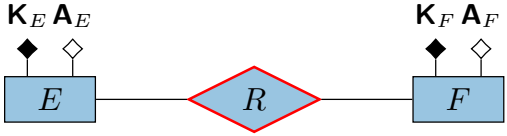
```
mappingId MAddress
source    SELECT client, locCity, locStreet FROM TAddress
target    :C/{client} :address :L/{locCity}/{locStreet} .
```

Mapping pattern: Relationship (MpR) – General case

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}})$



Mapping assertion:

$s : T_R$

$t : \texttt{:E/\{}\mathbf{K}_{RE}\texttt{\}}\ p_R\ \texttt{:F/\{}\mathbf{K}_{RF}\texttt{\}}\ .$

Ontology axioms:

$\exists p_R \sqsubseteq C_E$

$\exists p_R^- \sqsubseteq C_F$

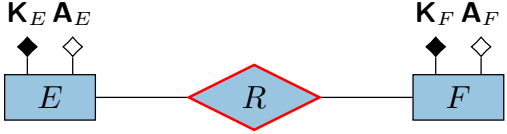
- In case of a $(1, n)$ cardinality on role R_E, the inclusion dependency $T_E[\mathbf{K}_E] \subseteq T_R[\mathbf{K}_{RE}]$ holds in the relational schema, and we add to the ontology the inclusion axiom $C_E \sqsubseteq \exists p_R$.
  Similarly for a cardinality $(1, n)$ on role R_F.
- In case of a $(\_, 1)$ cardinality on role R_E, the primary key for T_R is restricted to the attributes $\mathbf{K}_{RE}$. (Similarly for R_F and $\mathbf{K}_{RF}$.) In case both roles have a $(\_, 1)$ cardinality, either choice for the primary key is made, and the remaining attributes form a non-primary key in the logical schema.
- In case of a $(1, 1)$ cardinality on role R_E, both modifications above apply, and the inclusion dependency $T_E[\mathbf{K}_E] \subseteq T_R[\mathbf{K}_{RE}]$ is actually a foreign key. (Similarly for role R_F.)

Mapping pattern: Relationship (MpR) – General case

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}})$



Mapping assertion:

$s : T_R$

$t :$:E/{$\mathbf{K}_{RE}$} p_R :F/{$\mathbf{K}_{RF}$} .

Ontology axioms:

$\exists p_R \sqsubseteq C_E$

$\exists p_R^- \sqsubseteq C_F$

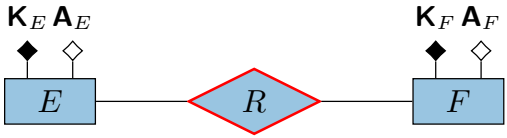
- In case of a $(1, n)$ cardinality on role R_E, the inclusion dependency $T_E[\mathbf{K}_E] \subseteq T_R[\mathbf{K}_{RE}]$ holds in the relational schema, and we add to the ontology the inclusion axiom $C_E \sqsubseteq \exists p_R$. Similarly for a cardinality $(1, n)$ on role R_F.
- In case of a $(\_, 1)$ cardinality on role R_E, the primary key for T_R is restricted to the attributes $\mathbf{K}_{RE}$. (Similarly for R_F and $\mathbf{K}_{RF}$.) In case both roles have a $(\_, 1)$ cardinality, either choice for the primary key is made, and the remaining attributes form a non-primary key in the logical schema.
- In case of a $(1, 1)$ cardinality on role R_E, both modifications above apply, and the inclusion dependency $T_E[\mathbf{K}_E] \subseteq T_R[\mathbf{K}_{RE}]$ is actually a foreign key. (Similarly for role R_F.)

Mapping pattern: Relationship (MpR) – General case

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $\quad$ $T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}})$



Mapping assertion:

$s : T_R$

$t :$:E/{$\mathbf{K}_{RE}$} p_R :F/{$\mathbf{K}_{RF}$} .

Ontology axioms:

$\exists p_R \sqsubseteq C_E$

$\exists p_R^- \sqsubseteq C_F$

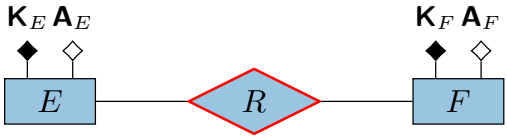
- In case of a $(1, n)$ cardinality on role R_E, the inclusion dependency $T_E[\mathbf{K}_E] \subseteq T_R[\mathbf{K}_{RE}]$ holds in the relational schema, and we add to the ontology the inclusion axiom $C_E \sqsubseteq \exists p_R$.
  Similarly for a cardinality $(1, n)$ on role R_F.
- In case of a $(\_, 1)$ cardinality on role R_E, the primary key for T_R is restricted to the attributes $\mathbf{K}_{RE}$. (Similarly for R_F and $\mathbf{K}_{RF}$.) In case both roles have a $(\_, 1)$ cardinality, either choice for the primary key is made, and the remaining attributes form a non-primary key in the logical schema.
- In case of a $(1, 1)$ cardinality on role R_E, both modifications above apply, and the inclusion dependency $T_E[\mathbf{K}_E] \subseteq T_R[\mathbf{K}_{RE}]$ is actually a foreign key. (Similarly for role R_F.)

Mapping pattern: Relationship (MpR) – General case

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $\quad$ $T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}})$



Mapping assertion:

$s : T_R$

$t :$:E/{$\mathbf{K}_{RE}$} p_R :F/{$\mathbf{K}_{RF}$} .

Ontology axioms:

$\exists p_R \sqsubseteq C_E$

$\exists p_R^- \sqsubseteq C_F$

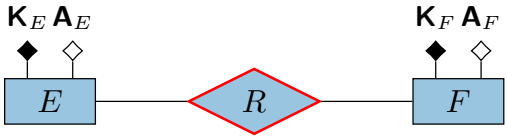
- In case of a $(1, n)$ cardinality on role R_E, the inclusion dependency $T_E[\mathbf{K}_E] \subseteq T_R[\mathbf{K}_{RE}]$ holds in the relational schema, and we add to the ontology the inclusion axiom $C_E \sqsubseteq \exists p_R$. Similarly for a cardinality $(1, n)$ on role R_F.
- In case of a $(\_, 1)$ cardinality on role R_E, the primary key for T_R is restricted to the attributes $\mathbf{K}_{RE}$. (Similarly for R_F and $\mathbf{K}_{RF}$.) In case both roles have a $(\_, 1)$ cardinality, either choice for the primary key is made, and the remaining attributes form a non-primary key in the logical schema.
- In case of a $(1, 1)$ cardinality on role R_E, both modifications above apply, and the inclusion dependency $T_E[\mathbf{K}_E] \subseteq T_R[\mathbf{K}_{RE}]$ is actually a foreign key. (Similarly for role R_F.)

Mapping pattern: Relationship (MpR) – General case

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $\quad$ $T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}})$



Mapping assertion:

$s : T_R$

$t :$:E/{$\mathbf{K}_{RE}$} p_R :F/{$\mathbf{K}_{RF}$} .

Ontology axioms:

$\exists p_R \sqsubseteq C_E$

$\exists p_R^- \sqsubseteq C_F$

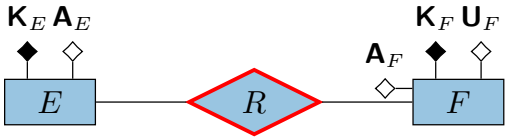
- In case of a $(1, n)$ cardinality on role R_E, the inclusion dependency $T_E[\mathbf{K}_E] \subseteq T_R[\mathbf{K}_{RE}]$ holds in the relational schema, and we add to the ontology the inclusion axiom $C_E \sqsubseteq \exists p_R$. Similarly for a cardinality $(1, n)$ on role R_F.
- In case of a $(\_, 1)$ cardinality on role R_E, the primary key for T_R is restricted to the attributes $\mathbf{K}_{RE}$. (Similarly for R_F and $\mathbf{K}_{RF}$.) In case both roles have a $(\_, 1)$ cardinality, either choice for the primary key is made, and the remaining attributes form a non-primary key in the logical schema.
- In case of a $(1, 1)$ cardinality on role R_E, both modifications above apply, and the inclusion dependency $T_E[\mathbf{K}_E] \subseteq T_R[\mathbf{K}_{RE}]$ is actually a foreign key. (Similarly for role R_F.)

Mapping pattern: Relationship with Identifier Alignment (MpRa)

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $\quad$ $T_F(\underline{\mathbf{K}_F}, \mathbf{U}_F, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}})$ $\quad$ $unique_{R_F}(\mathbf{U}_F)$



Mapping assertion:

$s : T_R \bowtie_{\mathbf{K}_{RF}=\mathbf{U}_F} T_F$

$t : \texttt{:E/\{}\mathbf{K}_{RE}\texttt{\}}\ p_R\ \texttt{:F/\{}\mathbf{K}_F\texttt{\}}\ .$

Ontology axioms:

$\exists p_R \sqsubseteq C_E$

$\exists p_R^- \sqsubseteq C_F$

For the application of the mapping pattern, we observe the following:

- Such pattern is a variation of pattern **MpR**, in which the foreign key in T_R does not point to the primary key $\mathbf{K}_F$ of T_F, but to an additional key $\mathbf{U}_F$.
- Since the instances of class C_F corresponding to T_F are constructed using the primary key $\mathbf{K}_F$ of T_F (cf. pattern **MpE**), also the pairs that populate p_R should refer in their object position to $\mathbf{K}_F$.
- Note that $\mathbf{K}_F$ can only be retrieved by a join between T_R and T_F on the additional key $\mathbf{U}_F$.

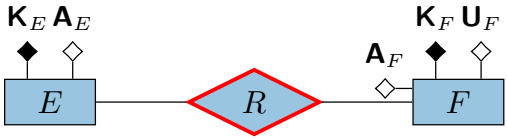
Cardinality constraints are handled similarly to **MpR**, with the difference that now the constraints involve $\mathbf{K}_{RF}$ and U_F. The case when both sets of attributes in T_R require alignment is treated similarly.

Mapping pattern: Relationship with Identifier Alignment (MpRa)

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $\quad$ $T_F(\underline{\mathbf{K}_F}, \mathbf{U}_F, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}})$ $\quad$ $unique_{R_F}(\mathbf{U}_F)$



Mapping assertion:

$s : T_R \bowtie_{\mathbf{K}_{RF}=\mathbf{U}_F} T_F$

$t :$ `:E/{`$\mathbf{K}_{RE}$`}` p_R `:F/{`$\mathbf{K}_F$`}` .

Ontology axioms:

$\exists p_R \sqsubseteq C_E$

$\exists p_R^- \sqsubseteq C_F$

For the application of the mapping pattern, we observe the following:

- Such pattern is a variation of pattern **MpR**, in which the foreign key in T_R does not point to the primary key $\mathbf{K}_F$ of T_F, but to an additional key $\mathbf{U}_F$.
- Since the instances of class C_F corresponding to T_F are constructed using the primary key $\mathbf{K}_F$ of T_F (cf. pattern **MpE**), also the pairs that populate p_R should refer in their object position to $\mathbf{K}_F$.
- Note that $\mathbf{K}_F$ can only be retrieved by a join between T_R and T_F on the additional key $\mathbf{U}_F$.

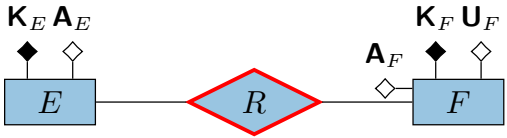
Cardinality constraints are handled similarly to **MpR**, with the difference that now the constraints involve $\mathbf{K}_{RF}$ and $\mathbf{U}_F$. The case when both sets of attributes in T_R require alignment is treated similarly.

Mapping pattern: Relationship with Identifier Alignment (MpRa)

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $\quad$ $T_F(\underline{\mathbf{K}_F}, \mathbf{U}_F, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}})$ $\quad$ $unique_{R_F}(\mathbf{U}_F)$



Mapping assertion:

$s : T_R \bowtie_{\mathbf{K}_{RF}=\mathbf{U}_F} T_F$

$t :$ `:E/{`$\mathbf{K}_{RE}$`}` p_R `:F/{`$\mathbf{K}_F$`}` .

Ontology axioms:

$\exists p_R \sqsubseteq C_E$

$\exists p_R^- \sqsubseteq C_F$

For the application of the mapping pattern, we observe the following:

- Such pattern is a variation of pattern **MpR**, in which the foreign key in T_R does not point to the primary key $\mathbf{K}_F$ of T_F, but to an additional key $\mathbf{U}_F$.
- Since the instances of class C_F corresponding to T_F are constructed using the primary key $\mathbf{K}_F$ of T_F (cf. pattern **MpE**), also the pairs that populate p_R should refer in their object position to $\mathbf{K}_F$.
- Note that $\mathbf{K}_F$ can only be retrieved by a join between T_R and T_F on the additional key $\mathbf{U}_F$.

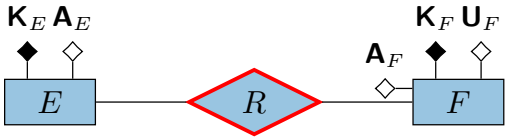
Cardinality constraints are handled similarly to **MpR**, with the difference that now the constraints involve $\mathbf{K}_{RF}$ and U_F. The case when both sets of attributes in T_R require alignment is treated similarly.

Mapping pattern: Relationship with Identifier Alignment (MpRa)

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $\quad$ $T_F(\underline{\mathbf{K}_F}, \mathbf{U}_F, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}})$ $\quad$ $unique_{R_F}(\mathbf{U}_F)$



Mapping assertion:

$s : T_R \bowtie_{\mathbf{K}_{RF}=\mathbf{U}_F} T_F$

$t :$ `:E/{`$\mathbf{K}_{RE}$`}` p_R `:F/{`$\mathbf{K}_F$`}` .

Ontology axioms:

$\exists p_R \sqsubseteq C_E$

$\exists p_R^- \sqsubseteq C_F$

For the application of the mapping pattern, we observe the following:

- Such pattern is a variation of pattern **MpR**, in which the foreign key in T_R does not point to the primary key $\mathbf{K}_F$ of T_F, but to an additional key $\mathbf{U}_F$.
- Since the instances of class C_F corresponding to T_F are constructed using the primary key $\mathbf{K}_F$ of T_F (cf. pattern **MpE**), also the pairs that populate p_R should refer in their object position to $\mathbf{K}_F$.
- Note that $\mathbf{K}_F$ can only be retrieved by a join between T_R and T_F on the additional key $\mathbf{U}_F$.

Cardinality constraints are handled similarly to **MpR**, with the difference that now the constraints involve $\mathbf{K}_{RF}$ and U_F. The case when both sets of attributes in T_R require alignment is treated similarly.

Mapping pattern: Rel. with Identifier Alignment (MpRa) – Example

The primary key of the table TLocationCoord is now not given by the city and street, which are used in the table TAddress that relates clients to their addresses, but is given by the latitude and longitude of locations.

```
TClient(ssn,name,dateOfBirth,hobbies)
TLocationCoord(latitude,longitude,city,street)    unique[TLocationCoord]: city,street
TAddress(client,locCity,locStreet)
    FK: TAddress[client] -> Tclient[ssn]
    FK: TAddress[locCity,locStreet] -> TLocationCoord[city,street]
```

Mapping: The TAddress table is mapped to an address object property, for which the ontology asserts that the domain is the class Client and the range an additional class Location, corresponding to the TLocationCoord table.

```
mappingId MAddressCoord
source    SELECT client, latitude, longitude
          FROM TAddress JOIN TLocationCoord ON locCity = city AND locStreet = street
target    :C/{client} :address :LC/{latitude}/{longitude} .
```

Mapping pattern: Rel. with Identifier Alignment (MpRa) – Example

The primary key of the table TLocationCoord is now not given by the city and street, which are used in the table TAddress that relates clients to their addresses, but is given by the latitude and longitude of locations.

```
TClient(ssn,name,dateOfBirth,hobbies)
TLocationCoord(latitude,longitude,city,street)    unique[TLocationCoord]: city,street
TAddress(client,locCity,locStreet)
    FK: TAddress[client] -> Tclient[ssn]
    FK: TAddress[locCity,locStreet] -> TLocationCoord[city,street]
```

Mapping: The TAddress table is mapped to an address object property, for which the ontology asserts that the domain is the class Client and the range an additional class Location, corresponding to the TLocationCoord table.

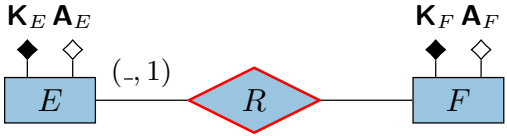
```
mappingId  MAddressCoord
source     SELECT client, latitude, longitude
           FROM TAddress JOIN TLocationCoord ON locCity = city AND locStreet = street
target     :C/{client} :address :LC/{latitude}/{longitude} .
```

Mapping pattern: Relationship with Merging (MpRm)

Relational schema and constraints:

$T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_E(\underline{\mathbf{K}_E}, \mathbf{K}_{EF}, \mathbf{A}_E)$



Mapping assertion:

$s : T_E$

$t :$ `:E/{`$\mathbf{K}_E$`}` p_{EF} `:F/{`$\mathbf{K}_{EF}$`}` .

Ontology axioms:

$\exists p_{EF} \sqsubseteq C_E$

$\exists p_{EF}^- \sqsubseteq C_F$

For the application of the mapping pattern, we observe the following:

- Such pattern is characterized by a table T_E in which the foreign key $\mathbf{K}_{EF}$ to a table T_F is disjoint from its primary key $\mathbf{K}_E$.
- The table T_E is mapped to an object property p_{EF}, whose subject and object are derived respectively from $\mathbf{K}_E$ and $\mathbf{K}_{EF}$.

Cardinality constraints are handled similarly to **MpR**, with the catch that in the case of $(0, 1)$ cardinality on role R_E, we have that $\mathbf{K}_{EF}$ is nullable.

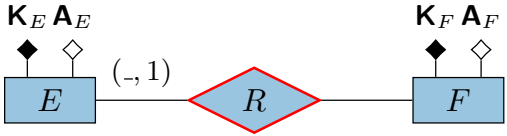
The alignment variant **MpRma**, where the foreign key $\mathbf{K}_{EF}$ of T_E references a non-primary identifier of T_F, is defined in the straightforward way.

Mapping pattern: Relationship with Merging (MpRm)

Relational schema and constraints:

$T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_E(\underline{\mathbf{K}_E}, \mathbf{K}_{EF}, \mathbf{A}_E)$



Mapping assertion:

$s : T_E$

$t :$ `:E/{`$\mathbf{K}_E$`}` p_{EF} `:F/{`$\mathbf{K}_{EF}$`}` .

Ontology axioms:

$\exists p_{EF} \sqsubseteq C_E$

$\exists p_{EF}^- \sqsubseteq C_F$

For the application of the mapping pattern, we observe the following:

- Such pattern is characterized by a table T_E in which the foreign key $\mathbf{K}_{EF}$ to a table T_F is disjoint from its primary key $\mathbf{K}_E$.
- The table T_E is mapped to an object property p_{EF}, whose subject and object are derived respectively from $\mathbf{K}_E$ and $\mathbf{K}_{EF}$.

Cardinality constraints are handled similarly to **MpR**, with the catch that in the case of $(0, 1)$ cardinality on role R_E, we have that $\mathbf{K}_{EF}$ is nullable.

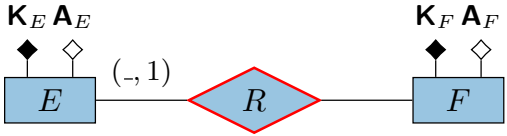
The alignment variant **MpRma**, where the foreign key $\mathbf{K}_{EF}$ of T_E references a non-primary identifier of T_F, is defined in the straightforward way.

Mapping pattern: Relationship with Merging (MpRm)

Relational schema and constraints:

$T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_E(\underline{\mathbf{K}_E}, \mathbf{K}_{EF}, \mathbf{A}_E)$



Mapping assertion:

$s : T_E$

$t :$ `:E/{`$\mathbf{K}_E$`}` p_{EF} `:F/{`$\mathbf{K}_{EF}$`}` .

Ontology axioms:

$\exists p_{EF} \sqsubseteq C_E$

$\exists p_{EF}^- \sqsubseteq C_F$

For the application of the mapping pattern, we observe the following:

- Such pattern is characterized by a table T_E in which the foreign key $\mathbf{K}_{EF}$ to a table T_F is disjoint from its primary key $\mathbf{K}_E$.
- The table T_E is mapped to an object property p_{EF}, whose subject and object are derived respectively from $\mathbf{K}_E$ and $\mathbf{K}_{EF}$.

Cardinality constraints are handled similarly to **MpR**, with the catch that in the case of $(0, 1)$ cardinality on role R_E, we have that $\mathbf{K}_{EF}$ is nullable.

The alignment variant **MpRma**, where the foreign key $\mathbf{K}_{EF}$ of T_E references a non-primary identifier of T_F, is defined in the straightforward way.

Mapping pattern: Relationship with Merging (MpRm) – Example

The relationship between a client and its unique billing address has been merged into the TClient table. The ontology defines a billingAddress object property, whose domain is the Client class and whose range is the Location class.

```
TLocation(city,street)
TClient(ssn,name,dateOfBirth,billCity,billStreet,hobbies)
    FK: TClient[billCity,billStreet] -> TLocation[city,street]
```

Mapping: The billing address information is extracted by a mapping from the TClient table to billingAddress.

```
mappingId MBillingAddress
source    SELECT ssn, billCity, billStreet FROM TClient
target    :C/{ssn} :billingAddress :L/{billCity}/{billStreet} .
```

Mapping pattern: Relationship with Merging (MpRm) – Example

The relationship between a client and its unique billing address has been merged into the TClient table. The ontology defines a billingAddress object property, whose domain is the Client class and whose range is the Location class.

```
TLocation(city,street)
TClient(ssn,name,dateOfBirth,billCity,billStreet,hobbies)
    FK: TClient[billCity,billStreet] -> TLocation[city,street]
```

Mapping: The billing address information is extracted by a mapping from the TClient table to billingAddress.

```
mappingId MBillingAddress
source    SELECT ssn, billCity, billStreet FROM TClient
target    :C/{ssn} :billingAddress :L/{billCity}/{billStreet} .
```

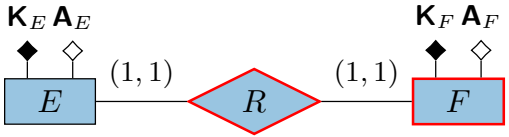
Mapping pattern: 1-1 Relationship with Merging (MpR11m)

Relational schema and constraints:

$$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E, \mathbf{K}_F, \mathbf{A}_F) \qquad \mathit{unique}_{T_E}(\mathbf{K}_F)$$

$$V_E(\underline{\mathbf{K}_E}, \mathbf{A}_E) = \pi_{\mathbf{K}_E, \mathbf{A}_E}(T_E) \qquad V_F(\underline{\mathbf{K}_F}, \mathbf{A}_F) = \pi_{\mathbf{K}_F, \mathbf{A}_F}(T_E)$$

$$V_R(\underline{\mathbf{K}_E}, \mathbf{K}_F) = \pi_{\mathbf{K}_E, \mathbf{K}_F}(T_E)$$



Mapping assertion:

$s : T_E$

t : `:F/{`$\mathbf{K}_F$`} rdf:type` C_F .
$\{$ `:F/{`$\mathbf{K}_F$`}` d_A $\{A\}$. $\}_{A \in \mathbf{K}_F \cup \mathbf{A}_F}$
`:E/{`$\mathbf{K}_E$`}` p_R `:F/{`$\mathbf{K}_F$`}` .

Ontology axioms:

$$\exists p_R \equiv C_E \qquad \left\{ \begin{array}{l} \exists d_A \sqsubseteq C_F \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_F \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{K}_F \cup \mathbf{A}_F}$$

$$\exists p_R^- \equiv C_F$$

For the application of the mapping pattern, we observe the following:

- The pattern could be applied when a table T_E has a primary key $\mathbf{K}_E$ and an additional key $\mathbf{K}_F$.
- Moreover, domain knowledge of the ontology indicates that objects with IRI `:F/{`$\mathbf{K}_F$`}` are relevant in the domain, and that they have data properties that correspond to the attributes $\mathbf{A}_F$ of T_E.
- When this pattern is applied, the key $\mathbf{K}_F$ and the attributes $\mathbf{A}_F$, can be projected out from T_E, resulting in a view V_E to which further patterns can be applied, including **MpR11m** itself.

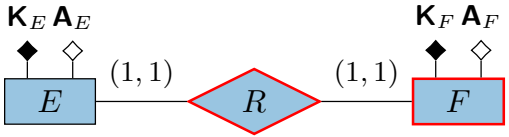
Mapping pattern: 1-1 Relationship with Merging (MpR11m)

Relational schema and constraints:

$$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E, \mathbf{K}_F, \mathbf{A}_F) \qquad \mathit{unique}_{T_E}(\mathbf{K}_F)$$

$$V_E(\underline{\mathbf{K}_E}, \mathbf{A}_E) = \pi_{\mathbf{K}_E, \mathbf{A}_E}(T_E) \qquad V_F(\underline{\mathbf{K}_F}, \mathbf{A}_F) = \pi_{\mathbf{K}_F, \mathbf{A}_F}(T_E)$$

$$V_R(\underline{\mathbf{K}_E}, \mathbf{K}_F) = \pi_{\mathbf{K}_E, \mathbf{K}_F}(T_E)$$



Mapping assertion:

$s : T_E$

t : :F/{$\mathbf{K}_F$} rdf:type C_F .
{ :F/{$\mathbf{K}_F$} d_A {A} . }$_{A \in \mathbf{K}_F \cup \mathbf{A}_F}$
:E/{$\mathbf{K}_E$} p_R :F/{$\mathbf{K}_F$} .

Ontology axioms:

$$\exists p_R \equiv C_E \qquad \left\{ \begin{array}{l} \exists d_A \sqsubseteq C_F \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_F \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{K}_F \cup \mathbf{A}_F}$$

$$\exists p_R^- \equiv C_F$$

For the application of the mapping pattern, we observe the following:

- The pattern could be applied when a table T_E has a primary key $\mathbf{K}_E$ and an additional key $\mathbf{K}_F$.
- Moreover, domain knowledge of the ontology indicates that objects with IRI :F/{$\mathbf{K}_F$} are relevant in the domain, and that they have data properties that correspond to the attributes $\mathbf{A}_F$ of T_E.
- When this pattern is applied, the key $\mathbf{K}_F$ and the attributes $\mathbf{A}_F$, can be projected out from T_E, resulting in a view V_E to which further patterns can be applied, including **MpR11m** itself.

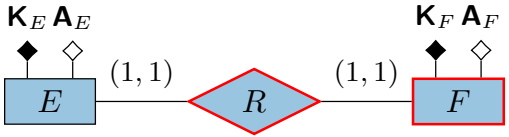
Mapping pattern: 1-1 Relationship with Merging (MpR11m)

Relational schema and constraints:

$$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E, \mathbf{K}_F, \mathbf{A}_F) \qquad \mathit{unique}_{T_E}(\mathbf{K}_F)$$

$$V_E(\underline{\mathbf{K}_E}, \mathbf{A}_E) = \pi_{\mathbf{K}_E, \mathbf{A}_E}(T_E) \qquad V_F(\underline{\mathbf{K}_F}, \mathbf{A}_F) = \pi_{\mathbf{K}_F, \mathbf{A}_F}(T_E)$$

$$V_R(\underline{\mathbf{K}_E}, \underline{\mathbf{K}_F}) = \pi_{\mathbf{K}_E, \mathbf{K}_F}(T_E)$$



Mapping assertion:

$s : T_E$

t : :F/{$\mathbf{K}_F$} rdf:type C_F .
{ :F/{$\mathbf{K}_F$} d_A {A} . }$_{A \in \mathbf{K}_F \cup \mathbf{A}_F}$
:E/{$\mathbf{K}_E$} p_R :F/{$\mathbf{K}_F$} .

Ontology axioms:

$$\exists p_R \equiv C_E \qquad \exists p_R^- \equiv C_F \qquad \left\{ \begin{array}{l} \exists d_A \sqsubseteq C_F \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_F \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{K}_F \cup \mathbf{A}_F}$$

For the application of the mapping pattern, we observe the following:

- The pattern could be applied when a table T_E has a primary key $\mathbf{K}_E$ and an additional key $\mathbf{K}_F$.
- Moreover, domain knowledge of the ontology indicates that objects with IRI :F/{$\mathbf{K}_F$} are relevant in the domain, and that they have data properties that correspond to the attributes $\mathbf{A}_F$ of T_E.
- When this pattern is applied, the key $\mathbf{K}_F$ and the attributes $\mathbf{A}_F$, can be projected out from T_E, resulting in a view V_E to which further patterns can be applied, including **MpR11m** itself.

Mapping pattern: 1-1 Relationship with Merging (MpR11m) – Example

A single table TUniversity, containing the information about universities, contains also information about their rector. The given ontology contains both a University and a Rector class.

```
TUniversity(uname,numfaculties,recssn,recname,recdob,salary)
    unique[TUniversity]: recssn
```

Mapping: The attribute recssn in TUniversity, identifying the rector, is used to form the IRIs for the instances of Rector, and the attributes recname and recdob, intuitively belonging to the rector, are mapped to data properties that have as domain Rector (as opposed to University).

```
mappingId MUniversity
source    SELECT uname, numfaculties FROM TUniversity
target    :U/{uname} rdf:type :University ;  :numfac {numfaculties} .

mappingId MRector
source    SELECT recssn, recname, recdob FROM TUniversity
target    :P/{recssn} rdf:type :Rector ;
                      :ssn {recssn} ;  :name {recname} ;  :dob {recdob} .

mappingId MhasRector
source    SELECT uname, recssn FROM TUniversity
target    :U/{uname} :hasRector :P/{recssn} .
```

Mapping pattern: 1-1 Relationship with Merging (MpR11m) – Example

A single table TUniversity, containing the information about universities, contains also information about their rector. The given ontology contains both a University and a Rector class.

```
TUniversity(uname,numfaculties,recssn,recname,recdob,salary)
    unique[TUniversity]: recssn
```

Mapping: The attribute recssn in TUniversity, identifying the rector, is used to form the IRIs for the instances of Rector, and the attributes recname and recdob, intuitively belonging to the rector, are mapped to data properties that have as domain Rector (as opposed to University).

```
mappingId MUniversity
source    SELECT uname, numfaculties FROM TUniversity
target    :U/{uname} rdf:type :University ;  :numfac {numfaculties} .

mappingId MRector
source    SELECT recssn, recname, recdob FROM TUniversity
target    :P/{recssn} rdf:type :Rector ;
                      :ssn {recssn} ;  :name {recname} ;  :dob {recdob} .

mappingId MhasRector
source    SELECT uname, recssn FROM TUniversity
target    :U/{uname} :hasRector :P/{recssn} .
```

Mapping pattern: 1-1 Relationship with Merging (MpR11m) – Notes

- Notice that to apply pattern **MpR11m**, domain knowledge is inherently required to determine to which class the attributes should be associated.
- For example, assume that the table TUniversity contains an attribute for the salary of the rector. Then, we have two possibilities:
  - the salary is considered a property of the rector, e.g., if the salary is negotiated individually by the rector.
  - the salary is considered a property of the university, e.g., if the salary of the rector is determined by some regulation of the university.

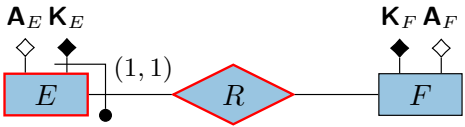
  Distinguishing which of these two possibilities is the correct one, requires in-depth knowledge about the domain.
- The necessary domain knowledge may also come from the ontology, e.g., if the data properties corresponding to the attributes are already present in the ontology, and their domain has been declared.

Mapping pattern: Entity with Weak Identification (MpEw)

Relational schema and constraints:

$T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_E(\underline{\mathbf{K}_E, \mathbf{K}_{EF}}, \mathbf{A}_E)$



Mapping assertions:

$s : T_E$

t : :E/{$\mathbf{K}_E$}/{$\mathbf{K}_{EF}$} rdf:type C_E .
{ :E/{$\mathbf{K}_E$}/{$\mathbf{K}_{EF}$} d_A {A} . }$_{A \in \mathbf{K}_E \cup \mathbf{A}_E}$
:E/{$\mathbf{K}_E$}/{$\mathbf{K}_{EF}$} p_R :F/{$\mathbf{K}_{EF}$} .

Ontology axioms:

$\exists p_R \equiv C_E$

$\exists p_R^- \sqsubseteq C_F$

$$\left\{ \begin{array}{l} \exists d_A \sqsubseteq C_E \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_E \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{K}_E \cup \mathbf{A}_E}$$

For the application of the mapping pattern, we observe the following:

- The data source contains a table T_E with primary key $\mathbf{K}_E, \mathbf{K}_{EF}$ and additional attributes $\mathbf{A}_E$.
- Attributes $\mathbf{K}_{EF}$ are a foreign key to an additional source table T_F. They are not to be mapped to data properties (for class C_E), since they act as external identifier for table T_E.
- The table T_F has a (primary) key $\mathbf{K}_F$ and may also contain additional attributes $\mathbf{A}_F$ (considered when applying **MpE** to it).
- The ontology contains an object property p_R corresponding to a relationship that has been merged into T_E, and classes C_E and C_F corresponding to T_E and T_F, respectively.

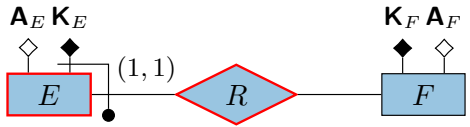
Cardinality constraints are handled similarly as for **MpR**. Optional attributes are handled similarly as for **MpE**.
The alignment variant **MpWEa**, where the foreign key references a non-primary identifier, is defined in the straightforward way.

Mapping pattern: Entity with Weak Identification (MpEw)

Relational schema and constraints:

$T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_E(\underline{\mathbf{K}_E, \mathbf{K}_{EF}}, \mathbf{A}_E)$



Mapping assertions:

$s : T_E$

t : :E/{$\mathbf{K}_E$}/{$\mathbf{K}_{EF}$} rdf:type C_E .
{ :E/{$\mathbf{K}_E$}/{$\mathbf{K}_{EF}$} d_A {A} . }$_{A \in \mathbf{K}_E \cup \mathbf{A}_E}$
:E/{$\mathbf{K}_E$}/{$\mathbf{K}_{EF}$} p_R :F/{$\mathbf{K}_{EF}$} .

Ontology axioms:

$\exists p_R \equiv C_E$

$\exists p_R^- \sqsubseteq C_F$

$$\left.\begin{array}{l} \exists d_A \sqsubseteq C_E \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_E \sqsubseteq \exists d_A \end{array}\right\}_{A \in \mathbf{K}_E \cup \mathbf{A}_E}$$

For the application of the mapping pattern, we observe the following:

- The data source contains a table T_E with primary key $\mathbf{K}_E, \mathbf{K}_{EF}$ and additional attributes $\mathbf{A}_E$.
- Attributes $\mathbf{K}_{EF}$ are a foreign key to an additional source table T_F. They are not to be mapped to data properties (for class C_E), since they act as external identifier for table T_E.
- The table T_F has a (primary) key $\mathbf{K}_F$ and may also contain additional attributes $\mathbf{A}_F$ (considered when applying **MpE** to it).
- The ontology contains an object property p_R corresponding to a relationship that has been merged into T_E, and classes C_E and C_F corresponding to T_E and T_F, respectively.

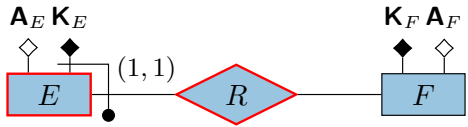
Cardinality constraints are handled similarly as for **MpR**. Optional attributes are handled similarly as for **MpE**.
The alignment variant **MpWEa**, where the foreign key references a non-primary identifier, is defined in the straightforward way.

Mapping pattern: Entity with Weak Identification (MpEw)

Relational schema and constraints:

$T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_E(\underline{\mathbf{K}_E, \mathbf{K}_{EF}}, \mathbf{A}_E)$



Mapping assertions:

$s : T_E$

t : :E/{$\mathbf{K}_E$}/{$\mathbf{K}_{EF}$} rdf:type C_E .
{ :E/{$\mathbf{K}_E$}/{$\mathbf{K}_{EF}$} d_A {A} . }$_{A \in \mathbf{K}_E \cup \mathbf{A}_E}$
:E/{$\mathbf{K}_E$}/{$\mathbf{K}_{EF}$} p_R :F/{$\mathbf{K}_{EF}$} .

Ontology axioms:

$\exists p_R \equiv C_E$

$\exists p_R^- \sqsubseteq C_F$

$$\left\{ \begin{array}{l} \exists d_A \sqsubseteq C_E \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_E \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{K}_E \cup \mathbf{A}_E}$$

For the application of the mapping pattern, we observe the following:

- The data source contains a table T_E with primary key $\underline{\mathbf{K}_E, \mathbf{K}_{EF}}$ and additional attributes $\mathbf{A}_E$.
- Attributes $\mathbf{K}_{EF}$ are a foreign key to an additional source table T_F. They are not to be mapped to data properties (for class C_E), since they act as external identifier for table T_E.
- The table T_F has a (primary) key $\underline{\mathbf{K}_F}$ and may also contain additional attributes $\mathbf{A}_F$ (considered when applying **MpE** to it).
- The ontology contains an object property p_R corresponding to a relationship that has been merged into T_E, and classes C_E and C_F corresponding to T_E and T_F, respectively.

Cardinality constraints are handled similarly as for **MpR**. Optional attributes are handled similarly as for **MpE**.
The alignment variant **MpWEa**, where the foreign key references a non-primary identifier, is defined in the straightforward way.

Mapping pattern: Entity with Weak Identification (MpEw) – Example

We consider two tables Student and University, and we are given an ontology that contains classes Student and University, connected through an object property enrollment.

```
TUniversity(uname,numfaculties)
TStudent(matrN,university,name)
    FK: TStudent[university] -> TUniversity[uname]
```

Mapping: The attributes matrN and university in TStudent, identifying the student, are used to form the IRIs for the instances of Student. These are put into correspondence with the University through the object property enrollment.

```
mappingId MStudent
source    SELECT matrN, name FROM TStudent
target    :S/{university}/{matrN} rdf:type :Student ;
                                  :matrN {matrN} ;  :name {name} .

mappingId Menrollment
source    SELECT matrN, university FROM TStudent
target    :S/{university}/{matrN} :enrollment :U/{university} .
```

Mapping pattern: Entity with Weak Identification (MpEw) – Example

We consider two tables Student and University, and we are given an ontology that contains classes Student and University, connected through an object property enrollment.

```
TUniversity(uname,numfaculties)
TStudent(matrN,university,name)
    FK: TStudent[university] -> TUniversity[uname]
```

Mapping: The attributes matrN and university in TStudent, identifying the student, are used to form the IRIs for the instances of Student. These are put into correspondence with the University through the object property enrollment.

```
mappingId  MStudent
source     SELECT matrN, name FROM TStudent
target     :S/{university}/{matrN} rdf:type :Student ;
                                   :matrN {matrN} ;  :name {name} .

mappingId  Menrollment
source     SELECT matrN, university FROM TStudent
target     :S/{university}/{matrN} :enrollment :U/{university} .
```

Associating properties to a property

OWL 2 QL does not allow one to assign data properties to an object property.

Example

Consider again the actsIn object property that relates MovieActors to Movies.
We might want to model in the ontology:

- the role in which the actor played in the movie;
- the duration of the appearance;
- the payment received for playing in the movie;
- ...

These are neither properties of an actor nor properties of a movie, but are properties related to the relationship between MovieActor and Movie.

We can take into account such situations by transforming an object property into a class, so that we can then attach the properties to the class.
This transformation is called **reification**, and follows a standard pattern.

Associating properties to a property

OWL 2 QL does not allow one to assign data properties to an object property.

Example

Consider again the actsIn object property that relates MovieActors to Movies.
We might want to model in the ontology:

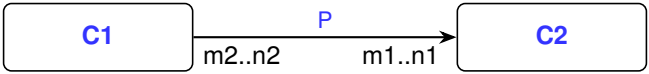
- the role in which the actor played in the movie;
- the duration of the appearance;
- the payment received for playing in the movie;
- ...

These are neither properties of an actor nor properties of a movie, but are properties related to the relationship between MovieActor and Movie.

We can take into account such situations by transforming an object property into a class, so that we can then attach the properties to the class.
This transformation is called **reification**, and follows a standard pattern.

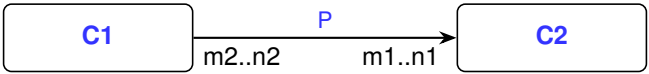
Reification of a property

Consider a property P with domain C_1 and range C_2, and suppose we want to associate (object or data) properties to P.

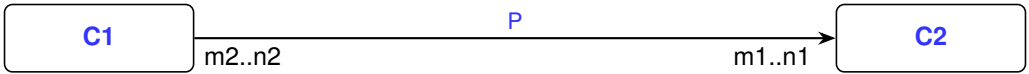


Reification of a property

Consider a property P with domain C_1 and range C_2, and suppose we want to associate (object or data) properties to P.

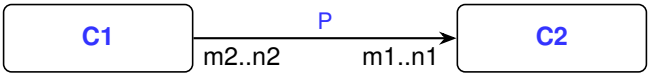


Reification of a property P with domain C_1 and range C_2



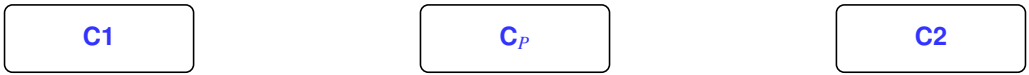
Reification of a property

Consider a property P with domain C_1 and range C_2, and suppose we want to associate (object or data) properties to P.



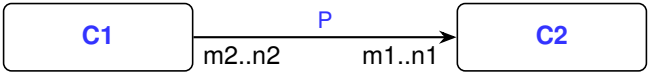
Reification of a property P with domain C_1 and range C_2

1. Introduce a new class C_P.



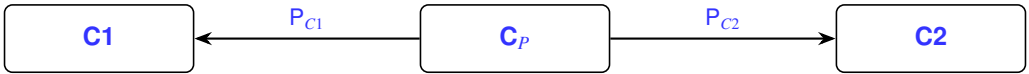
Reification of a property

Consider a property P with domain C_1 and range C_2, and suppose we want to associate (object or data) properties to P.



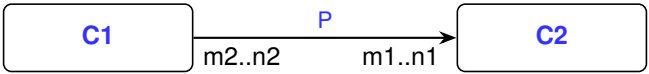
Reification of a property P with domain C_1 and range C_2

1. Introduce a new class C_P.
2. Introduce two new object properties, P_{C1}, connecting C_P to C_1, and P_{C2}, connecting C_P to C_2.



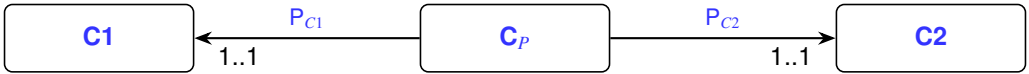
Reification of a property

Consider a property P with domain C_1 and range C_2, and suppose we want to associate (object or data) properties to P.



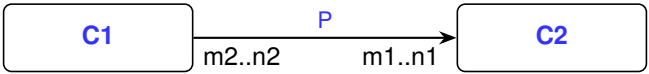
Reification of a property P with domain C_1 and range C_2

1. Introduce a new class C_P.
2. Introduce two new object properties, P_{C1}, connecting C_P to C_1, and P_{C2}, connecting C_P to C_2.
3. C_P has a mandatory and functional participation both to P_{C1} and to P_{C2}.



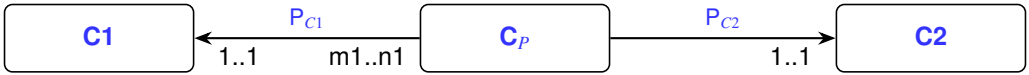
Reification of a property

Consider a property P with domain C_1 and range C_2, and suppose we want to associate (object or data) properties to P.



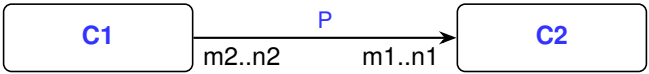
Reification of a property P with domain C_1 and range C_2

1. Introduce a new class C_P.
2. Introduce two new object properties, P_{C1}, connecting C_P to C_1, and P_{C2}, connecting C_P to C_2.
3. C_P has a mandatory and functional participation both to P_{C1} and to P_{C2}.
4. The cardinalities on P become cardinalities on P_{C1}^- (i.e., on the inverse of P_{C1}).



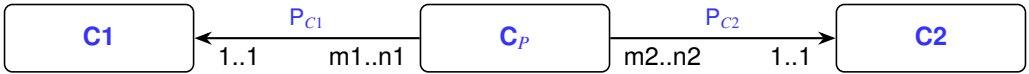
Reification of a property

Consider a property P with domain C_1 and range C_2, and suppose we want to associate (object or data) properties to P.



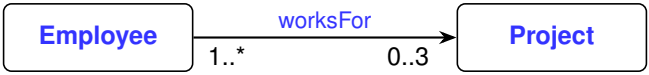
Reification of a property P with domain C_1 and range C_2

1. Introduce a new class C_P.
2. Introduce two new object properties, P_{C1}, connecting C_P to C_1, and P_{C2}, connecting C_P to C_2.
3. C_P has a mandatory and functional participation both to P_{C1} and to P_{C2}.
4. The cardinalities on P become cardinalities on P^-_{C1} (i.e., on the inverse of P_{C1}).
5. The cardinalities on P^- become cardinalities on P^-_{C2} (i.e., on the inverse of P_{C2}).



Reification of a property – Example

Consider the worksFor object property between the classes Employee and Project, expressing the fact that an employee works for a project, where each employee can work for at most three projects, and each project should have at least one employee working for it.

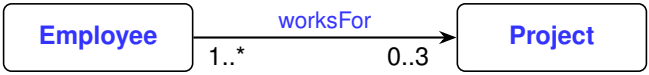


Suppose that we want to model also the dates when the employee started and ended her work for the project, and the number of person months she dedicated to that work. To do so, we need to reify the worksFor data property.

We introduce a class Work, which is the reified counterpart of worksFor, and connect it to Employee via a new object property workBy, and to Project via a new object property workFor.

Reification of a property – Example

Consider the worksFor object property between the classes Employee and Project, expressing the fact that an employee works for a project, where each employee can work for at most three projects, and each project should have at least one employee working for it.

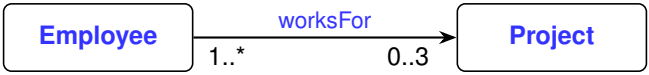


Suppose that we want to model also the dates when the employee started and ended her work for the project, and the number of person months she dedicated to that work. To do so, we need to reify the worksFor data property.

We introduce a class Work, which is the reified counterpart of worksFor, and connect it to Employee via a new object property workBy, and to Project via a new object property workFor.

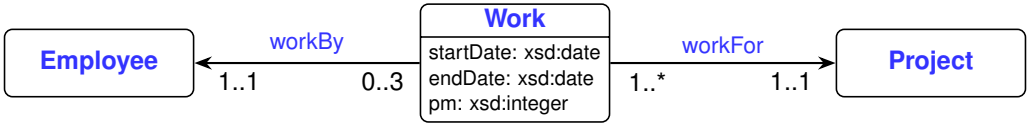
Reification of a property – Example

Consider the worksFor object property between the classes Employee and Project, expressing the fact that an employee works for a project, where each employee can work for at most three projects, and each project should have at least one employee working for it.



Suppose that we want to model also the dates when the employee started and ended her work for the project, and the number of person months she dedicated to that work. To do so, we need to reify the worksFor data property.

We introduce a class Work, which is the reified counterpart of worksFor, and connect it to Employee via a new object property workBy, and to Project via a new object property workFor.

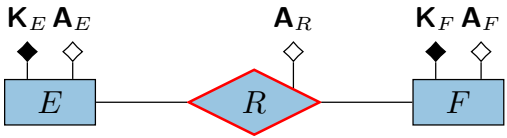


Mapping pattern: Reified Relationship (MpRR) – Attribute case

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $\quad$ $T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}}, \mathbf{A}_R)$



Mapping assertion:

$s : T_R$

t : :R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$} rdf:type C_R .
{ :R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$} d_A {A} . }$_{A \in \mathbf{K}_{RE} \cup \mathbf{K}_{RF} \cup \mathbf{A}_R}$
:R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$} p_{RE} :E/{$\mathbf{K}_{RE}$} .
:R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$} p_{RF} :F/{$\mathbf{K}_{RF}$} .

Ontology axioms:

$\exists p_{RE} \sqsubseteq C_R$ $\quad$ $\exists p_{RF} \sqsubseteq C_R$

$\exists p_{RE}^- \sqsubseteq C_E$ $\quad$ $\exists p_{RF}^- \sqsubseteq C_F$

$$\left\{ \begin{array}{l} \exists d_A \sqsubseteq C_R \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_R \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{A}_R}$$

For the application of the mapping pattern, we observe the following:

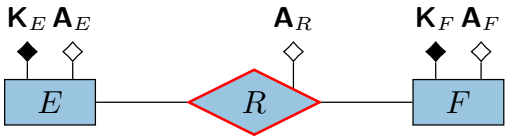
- The pattern applies to a table T_R whose primary key is partitioned in (at least) two parts $\mathbf{K}_{RE}$ and $\mathbf{K}_{RF}$ that are foreign keys to additional tables, and there are additional attributes $\mathbf{A}_R$ in T_R.
- Since T_R corresponds to a conceptual element that has itself properties (corresponding to $\mathbf{A}_R$), to represent it in the ontology we require a class C_R whose instances have an IRI :R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$}.
- The mapping ensures that each component of the relationship is represented by an object property (p_{RE}, p_{RF}), and that the tuples instantiating them can all be derived from T_R alone.

Mapping pattern: Reified Relationship (MpRR) – Attribute case

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $\quad$ $T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}}, \underline{\mathbf{K}_{RF}}, \mathbf{A}_R)$



Mapping assertion:

$s : T_R$

t : :R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$} rdf:type C_R .
{ :R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$} d_A {A} . }$_{A \in \mathbf{K}_{RE} \cup \mathbf{K}_{RF} \cup \mathbf{A}_R}$
:R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$} p_{RE} :E/{$\mathbf{K}_{RE}$} .
:R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$} p_{RF} :F/{$\mathbf{K}_{RF}$} .

Ontology axioms:

$\exists p_{RE} \sqsubseteq C_R$ $\quad$ $\exists p_{RF} \sqsubseteq C_R$

$\exists p_{RE}^- \sqsubseteq C_E$ $\quad$ $\exists p_{RF}^- \sqsubseteq C_F$

$$\left\{ \begin{array}{l} \exists d_A \sqsubseteq C_R \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_R \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{A}_R}$$

For the application of the mapping pattern, we observe the following:

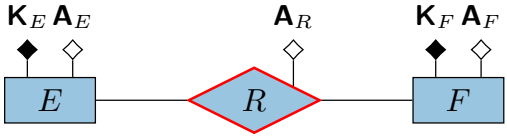
- The pattern applies to a table T_R whose primary key is partitioned in (at least) two parts $\mathbf{K}_{RE}$ and $\mathbf{K}_{RF}$ that are foreign keys to additional tables, and there are additional attributes $\mathbf{A}_R$ in T_R.
- Since T_R corresponds to a conceptual element that has itself properties (corresponding to $\mathbf{A}_R$), to represent it in the ontology we require a class C_R whose instances have an IRI :R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$}.
- The mapping ensures that each component of the relationship is represented by an object property (p_{RE}, p_{RF}), and that the tuples instantiating them can all be derived from T_R alone.

Mapping pattern: Reified Relationship (MpRR) – Attribute case

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $\quad$ $T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}}, \mathbf{A}_R)$



Mapping assertion:

$s : T_R$

t : :R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$} rdf:type C_R .
{ :R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$} d_A {A} . }$_{A \in \mathbf{K}_{RE} \cup \mathbf{K}_{RF} \cup \mathbf{A}_R}$
:R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$} p_{RE} :E/{$\mathbf{K}_{RE}$} .
:R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$} p_{RF} :F/{$\mathbf{K}_{RF}$} .

Ontology axioms:

$\exists p_{RE} \sqsubseteq C_R$ $\quad$ $\exists p_{RF} \sqsubseteq C_R$

$\exists p_{RE}^- \sqsubseteq C_E$ $\quad$ $\exists p_{RF}^- \sqsubseteq C_F$

$$\left\{ \begin{array}{l} \exists d_A \sqsubseteq C_R \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_R \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{A}_R}$$

For the application of the mapping pattern, we observe the following:

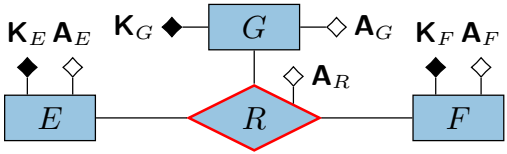
- The pattern applies to a table T_R whose primary key is partitioned in (at least) two parts $\mathbf{K}_{RE}$ and $\mathbf{K}_{RF}$ that are foreign keys to additional tables, and there are additional attributes $\mathbf{A}_R$ in T_R.
- Since T_R corresponds to a conceptual element that has itself properties (corresponding to $\mathbf{A}_R$), to represent it in the ontology we require a class C_R whose instances have an IRI :R/{$\mathbf{K}_{RE}$}/{$\mathbf{K}_{RF}$}.
- The mapping ensures that each component of the relationship is represented by an object property (p_{RE}, p_{RF}), and that the tuples instantiating them can all be derived from T_R alone.

Mapping pattern: Reified Relationship (MpRR) – n-ary relationship case

Relational schema and constraints:

$$T_G(\underline{\mathbf{K}_G}, \mathbf{A}_G)$$

$$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E) \quad T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}, \mathbf{K}_{RG}}, \mathbf{A}_R) \quad T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$$



Mapping assertion: $\mathbf{K}_R := \mathbf{K}_{RE} \cup \mathbf{K}_{RF} \cup \mathbf{K}_{RG}$

$s : T_R$

$t :$:R/{$\mathbf{K}_R$} rdf:type C_R .
{ :R/{$\mathbf{K}_R$} d_A {A} . }$_{A \in \mathbf{K}_R \cup \mathbf{A}_R}$
:R/{$\mathbf{K}_R$} p_{RE} :E/{$\mathbf{K}_{RE}$} .
:R/{$\mathbf{K}_R$} p_{RF} :F/{$\mathbf{K}_{RF}$} . :R/{$\mathbf{K}_R$} p_{RG} :G/{$\mathbf{K}_{RG}$} .

Ontology axioms:

$$\exists p_{RE} \sqsubseteq C_R \qquad \exists p_{RF} \sqsubseteq C_R \qquad \exists p_{RG} \sqsubseteq C_R$$

$$\exists p_{RE}^- \sqsubseteq C_E \qquad \exists p_{RF}^- \sqsubseteq C_F \qquad \exists p_{RG}^- \sqsubseteq C_G$$

$$\left\{ \begin{array}{l} \exists d_A \sqsubseteq C_R \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_R \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{A}_R}$$

For the application of the mapping pattern, we observe the following:

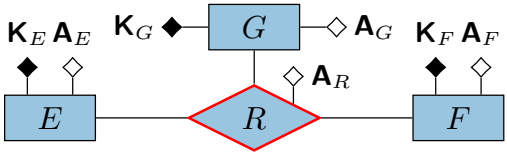
- The pattern applies to a table T_R whose primary key is partitioned in at least three parts $\mathbf{K}_{RE}$, $\mathbf{K}_{RF}$, and $\mathbf{K}_{RG}$, that are foreign keys to three additional tables.
- Additional attributes $\mathbf{A}_R$ might also be present in T_R.
- Apart from the arity of the relationship, the pattern behaves analogously to **MpRR** for the attribute case.

Mapping pattern: Reified Relationship (MpRR) – n-ary relationship case

Relational schema and constraints:

$$T_G(\underline{\mathbf{K}_G}, \mathbf{A}_G)$$

$$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E) \quad T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}, \mathbf{K}_{RG}}, \mathbf{A}_R) \quad T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$$



Mapping assertion: $\mathbf{K}_R := \mathbf{K}_{RE} \cup \mathbf{K}_{RF} \cup \mathbf{K}_{RG}$

$s : T_R$

t : :R/{$\mathbf{K}_R$} rdf:type C_R .
{ :R/{$\mathbf{K}_R$} d_A {A} . }$_{A \in \mathbf{K}_R \cup \mathbf{A}_R}$
:R/{$\mathbf{K}_R$} p_{RE} :E/{$\mathbf{K}_{RE}$} .
:R/{$\mathbf{K}_R$} p_{RF} :F/{$\mathbf{K}_{RF}$} . :R/{$\mathbf{K}_R$} p_{RG} :G/{$\mathbf{K}_{RG}$} .

Ontology axioms:

$$\exists p_{RE} \sqsubseteq C_R \qquad \exists p_{RF} \sqsubseteq C_R \qquad \exists p_{RG} \sqsubseteq C_R$$
$$\exists p_{RE}^- \sqsubseteq C_E \qquad \exists p_{RF}^- \sqsubseteq C_F \qquad \exists p_{RG}^- \sqsubseteq C_G$$
$$\left\{ \begin{array}{l} \exists d_A \sqsubseteq C_R \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_R \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{A}_R}$$

For the application of the mapping pattern, we observe the following:

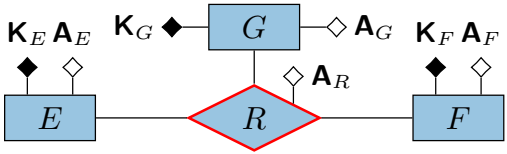
- The pattern applies to a table T_R whose primary key is partitioned in at least three parts $\mathbf{K}_{RE}$, $\mathbf{K}_{RF}$, and $\mathbf{K}_{RG}$, that are foreign keys to three additional tables.
- Additional attributes $\mathbf{A}_R$ might also be present in T_R.
- Apart from the arity of the relationship, the pattern behaves analogously to **MpRR** for the attribute case.

Mapping pattern: Reified Relationship (MpRR) – n-ary relationship case

Relational schema and constraints:

$$T_G(\underline{\mathbf{K}_G}, \mathbf{A}_G)$$

$$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E) \quad T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}, \mathbf{K}_{RG}}, \mathbf{A}_R) \quad T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$$



Mapping assertion: $\mathbf{K}_R := \mathbf{K}_{RE} \cup \mathbf{K}_{RF} \cup \mathbf{K}_{RG}$

$s : T_R$

t : :R/{$\mathbf{K}_R$} rdf:type C_R .
{ :R/{$\mathbf{K}_R$} d_A {A} . }$_{A \in \mathbf{K}_R \cup \mathbf{A}_R}$
:R/{$\mathbf{K}_R$} p_{RE} :E/{$\mathbf{K}_{RE}$} .
:R/{$\mathbf{K}_R$} p_{RF} :F/{$\mathbf{K}_{RF}$} . :R/{$\mathbf{K}_R$} p_{RG} :G/{$\mathbf{K}_{RG}$} .

Ontology axioms:

$$\exists p_{RE} \sqsubseteq C_R \qquad \exists p_{RF} \sqsubseteq C_R \qquad \exists p_{RG} \sqsubseteq C_R$$
$$\exists p_{RE}^- \sqsubseteq C_E \qquad \exists p_{RF}^- \sqsubseteq C_F \qquad \exists p_{RG}^- \sqsubseteq C_G$$
$$\left\{ \begin{array}{l} \exists d_A \sqsubseteq C_R \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_R \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{A}_R}$$

For the application of the mapping pattern, we observe the following:

- The pattern applies to a table T_R whose primary key is partitioned in at least three parts $\mathbf{K}_{RE}$, $\mathbf{K}_{RF}$, and $\mathbf{K}_{RG}$, that are foreign keys to three additional tables.
- Additional attributes $\mathbf{A}_R$ might also be present in T_R.
- Apart from the arity of the relationship, the pattern behaves analogously to **MpRR** for the attribute case.

Mapping pattern: Reified Relationship (MpRR) – Example

Consider a table TExam containing information about university exams, (which involve a student, a course, and a professor teaching that course), that has foreign keys towards three tables, namely TStudent, TCourse, and TProfessor.

```
TExam(student,course,professor,grade)
TStudent(ssn,sname)
TCourse(cid,cname,credits)
TProfessor(ssn,pname,level)
```

```
FK: TExam[student] -> TStudent[ssn]
FK: TExam[course] -> TCourse[cid]
FK: TExam[professor] -> TProfessor[ssn]
```

Mapping: This information is represented by a relationship that is inherently ternary. The ontology should contain a class Exam corresponding to the reified relationship, connected via three object properties to the classes Student, Course, and Professor. The mapping ensures that the class Exam is instantiated with objects whose IRI is constructed from the identifiers of the component classes.

```
mappingId MExam
source    SELECT student, course, professor, grade FROM TExam
target    :E/{student}/{course}/{professor} rdf:type :Exam ;
                                            :examOf :P/{student} ;
                                            :examFor :C/{course} ;
                                            :examBy :P/{professor} ;
                                            :examGrade {grade} .
```

Mapping pattern: Reified Relationship (MpRR) – Example

Consider a table TExam containing information about university exams, (which involve a student, a course, and a professor teaching that course), that has foreign keys towards three tables, namely TStudent, TCourse, and TProfessor.

```
TExam(student,course,professor,grade)
TStudent(ssn,sname)                    FK: TExam[student] -> TStudent[ssn]
TCourse(cid,cname,credits)             FK: TExam[course] -> TCourse[cid]
TProfessor(ssn,pname,level)            FK: TExam[professor] -> TProfessor[ssn]
```

Mapping: This information is represented by a relationship that is inherently ternary. The ontology should contain a class Exam corresponding to the reified relationship, connected via three object properties to the classes Student, Course, and Professor. The mapping ensures that the class Exam is instantiated with objects whose IRI is constructed from the identifiers of the component classes.

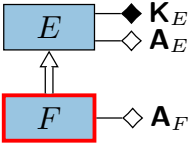
```
mappingId MExam
source    SELECT student, course, professor, grade FROM TExam
target    :E/{student}/{course}/{professor} rdf:type :Exam ;
                                            :examOf :P/{student} ;
                                            :examFor :C/{course} ;
                                            :examBy :P/{professor} ;
                                            :examGrade {grade} .
```

Mapping pattern: Hierarchy (MpH)

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$

$T_F(\underline{\mathbf{K}_{FE}}, \mathbf{A}_F)$



Mapping assertions:

$s : T_F$

t : :E/{$\mathbf{K}_{FE}$} rdf:type C_F .
{ :E/{$\mathbf{K}_{FE}$} d_A {A} . }$_{A \in \mathbf{A}_F}$

Ontology axioms:

$C_F \sqsubseteq C_E$

$$\left\{ \begin{array}{l} \exists d_A \sqsubseteq C_F \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_F \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{A}_F}$$

Optional attributes are handled as for **MpE**.

For the application of the mapping pattern, we observe the following:

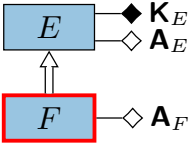
- The pattern considers a table T_F whose primary key is a foreign key to a table T_E.
- Then, T_F is mapped to a class C_F in the ontology that is a sub-class of the class C_E to which T_E is mapped.
- Hence, C_F "inherits" the template :E/{·} of C_E, so that the instances of the two classes are "compatible".

Mapping pattern: Hierarchy (MpH)

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$

$T_F(\underline{\mathbf{K}_{FE}}, \mathbf{A}_F)$ with $\mathbf{K}_{FE} \rightarrow \mathbf{K}_E$



Mapping assertions:

$s : T_F$

t : `:E/{`$\mathbf{K}_{FE}$`} rdf:type` C_F .
`{ :E/{`$\mathbf{K}_{FE}$`}` d_A `{`A`} . }`$_{A \in \mathbf{A}_F}$

Ontology axioms:

$$C_F \sqsubseteq C_E \qquad \left\{ \begin{array}{l} \exists d_A \sqsubseteq C_F \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_F \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{A}_F}$$

Optional attributes are handled as for **MpE**.

For the application of the mapping pattern, we observe the following:

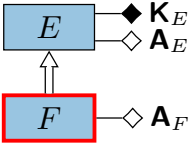
- The pattern considers a table T_F whose primary key is a foreign key to a table T_E.
- Then, T_F is mapped to a class C_F in the ontology that is a sub-class of the class C_E to which T_E is mapped.
- Hence, C_F "inherits" the template `:E/{·}` of C_E, so that the instances of the two classes are "compatible".

Mapping pattern: Hierarchy (MpH)

Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$

$T_F(\underline{\mathbf{K}_{FE}}, \mathbf{A}_F)$



Mapping assertions:

$s : T_F$

t : `:E/{`$\mathbf{K}_{FE}$`} rdf:type` C_F .

`{ :E/{`$\mathbf{K}_{FE}$`}` d_A `{A} . }`$_{A \in \mathbf{A}_F}$

Ontology axioms:

$$C_F \sqsubseteq C_E \qquad \left\{ \begin{array}{l} \exists d_A \sqsubseteq C_F \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_F \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{A}_F}$$

Optional attributes are handled as for **MpE**.

For the application of the mapping pattern, we observe the following:

- The pattern considers a table T_F whose primary key is a foreign key to a table T_E.
- Then, T_F is mapped to a class C_F in the ontology that is a sub-class of the class C_E to which T_E is mapped.
- Hence, C_F "inherits" the template `:E/{·}` of C_E, so that the instances of the two classes are "compatible".

Mapping pattern: Hierarchy (MpH) – Example

Consider a table TPerson containing information about persons, and a table TStudent containing information about students, which has a foreign key towards TPerson.

```
TPerson(ssn,name,dob)
TStudent(ssn,sid,credits)                FK: TStudent[ssn] -> TPerson[ssn]
```

Mapping: The two tables TPerson and TStudent are mapped to two classes Person and Student, respectively, each with data properties corresponding to the attributes of the table. Moreover, the ontology will contain an axiom stating that Student is a sub-class of Person.

```
mappingId MPerson
source    SELECT ssn, name, dob FROM TPerson
target    :P/{ssn} rdf:type :Person ;
                   :name {name} ;
                   :dateOfBirth {dob} .

mappingId MStudent
source    SELECT ssn, sid FROM TStudent
target    :P/{ssn} rdf:type :Student ;
                   :studentId {sid} ;
                   :hasCredits {credits} .
```

Mapping pattern: Hierarchy (MpH) – Example

Consider a table TPerson containing information about persons, and a table TStudent containing information about students, which has a foreign key towards TPerson.

```
TPerson(ssn,name,dob)
TStudent(ssn,sid,credits)                FK: TStudent[ssn] -> TPerson[ssn]
```

Mapping: The two tables TPerson and TStudent are mapped to two classes Person and Student, respectively, each with data properties corresponding to the attributes of the table. Moreover, the ontology will contain an axiom stating that Student is a sub-class of Person.

```
mappingId  MPerson
source     SELECT ssn, name, dob FROM TPerson
target     :P/{ssn} rdf:type :Person ;
                    :name {name} ;
                    :dateOfBirth {dob} .

mappingId  MStudent
source     SELECT ssn, sid FROM TStudent
target     :P/{ssn} rdf:type :Student ;
                    :studentId {sid} ;
                    :hasCredits {credits} .
```

Mapping pattern: Hierarchy with Identifier Alignment (MpHa)

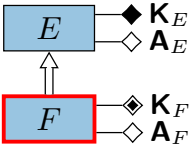
Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ *unique*$_{T_F}(\mathbf{U}_F)$

$T_F(\underline{\mathbf{K}_F}, \underline{\mathbf{U}_F}, \mathbf{A}_F)$

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ *unique*$_{V_F}(\mathbf{K}_F)$

$V_F(\mathbf{K}_F, \underline{\mathbf{U}_F}, \mathbf{A}_F) = T_F$



Mapping assertions:

$s : T_F$

$t : \texttt{:E/\{}\mathbf{U}_F\texttt{\}}\ \texttt{rdf:type}\ C_F\ .$

$\{\ \texttt{:E/\{}\mathbf{U}_F\texttt{\}}\ d_A\ \ \{A\}\ .\ \}_{A \in \mathbf{K}_F \cup \mathbf{A}_F}$

Ontology axioms:

$C_F \sqsubseteq C_E$ $\quad \left\{ \begin{array}{l} \exists d_A \sqsubseteq C_F \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_F \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{K}_F \cup \mathbf{A}_F}$

For the application of the mapping pattern, we observe the following:

- Such pattern is like **MpH**, but the foreign key in T_F is over a key $\mathbf{U}_F$ that is not primary.
- The objects for C_F have to be built out of $\mathbf{U}_F$, rather than out of its primary key $\mathbf{K}_F$.
- For this purpose, the pattern creates a view V_F in which $\mathbf{U}_F$ is the primary key, and the foreign key relations are preserved.

Mapping pattern: Hierarchy with Identifier Alignment (MpHa)

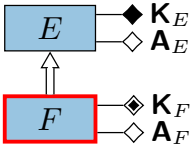
Relational schema and constraints:

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $\quad$ $unique_{T_F}(\mathbf{U}_F)$

$T_F(\underline{\mathbf{K}_F}, \underline{\mathbf{U}_F}, \mathbf{A}_F)$

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $\quad$ $unique_{V_F}(\mathbf{K}_F)$

$V_F(\mathbf{K}_F, \underline{\mathbf{U}_F}, \mathbf{A}_F) = T_F$



Mapping assertions:

$s : T_F$

t : :E/{$\mathbf{U}_F$} rdf:type C_F .

{ :E/{$\mathbf{U}_F$} d_A {A} . }$_{A \in \mathbf{K}_F \cup \mathbf{A}_F}$

Ontology axioms:

$$C_F \sqsubseteq C_E \qquad \left\{ \begin{array}{l} \exists d_A \sqsubseteq C_F \\ \exists d_A^- \sqsubseteq \mu(\tau(A)) \\ C_F \sqsubseteq \exists d_A \end{array} \right\}_{A \in \mathbf{K}_F \cup \mathbf{A}_F}$$

For the application of the mapping pattern, we observe the following:

- Such pattern is like **MpH**, but the foreign key in T_F is over a key $\mathbf{U}_F$ that is not primary.
- The objects for C_F have to be built out of $\mathbf{U}_F$, rather than out of its primary key $\mathbf{K}_F$.
- For this purpose, the pattern creates a view V_F in which $\mathbf{U}_F$ is the primary key, and the foreign key relations are preserved.

Mapping pattern: Hierarchy with Indentifier Alignment (MpHa) – Example

Consider the tables TPerson and TStudent of the previous example, but assume now that the primary key of TStudent is sid. Consider also an additional table TEnrolled, recording course enrollments.

```
TPerson(ssn,name,dob)
TStudent(sid,ssn,credits)      FK: TStudent[ssn] -> TPerson[ssn]     key[TStudent]: ssn
TEnrolled(student,course)      FK: TEnrolled[student] -> TStudent[sid]
```

Mapping: By applying pattern **MpHa**, we identify the instances of Student by their ssn, and we create a view `VStudent(sid,ssn,credits)`. But now, considering this view instead of TStudent, in order to map TEnrolled into an object property enrolledIn, we need to apply pattern **MpRa** rather than **MpR**.

```
mappingId MPerson
source    SELECT ssn, name, dob FROM TPerson
target    :P/{ssn} rdf:type :Person ;    :name {name} ;    :dateOfBirth {dob} .

mappingId MStudent
source    SELECT sid, ssn, credits FROM TStudent
target    :P/{ssn} rdf:type :Student ;   :studentId {sid} ;   :hasCredits {credits} .

mappingId MEnrolled
source    SELECT ssn, course FROM TEnrolled JOIN TStudent ON student = sid
target    :P/{ssn} :enrolledIn :C/{course} .
```

Mapping pattern: Hierarchy with Indentifier Alignment (MpHa) – Example

Consider the tables TPerson and TStudent of the previous example, but assume now that the primary key of TStudent is sid. Consider also an additional table TEnrolled, recording course enrollments.

```
TPerson(ssn,name,dob)
TStudent(sid,ssn,credits)      FK: TStudent[ssn] -> TPerson[ssn]     key[TStudent]: ssn
TEnrolled(student,course)      FK: TEnrolled[student] -> TStudent[sid]
```

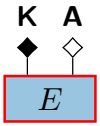
Mapping: By applying pattern **MpHa**, we identify the instances of Student by their ssn, and we create a view `VStudent(sid,ssn,credits)`. But now, considering this view instead of TStudent, in order to map TEnrolled into an object property enrolledIn, we need to apply pattern **MpRa** rather than **MpR**.

```
mappingId MPerson
source    SELECT ssn, name, dob FROM TPerson
target    :P/{ssn} rdf:type :Person ;    :name {name} ;    :dateOfBirth {dob} .

mappingId MStudent
source    SELECT sid, ssn, credits FROM TStudent
target    :P/{ssn} rdf:type :Student ;   :studentId {sid} ;   :hasCredits {credits} .

mappingId MEnrolled
source    SELECT ssn, course FROM TEnrolled JOIN TStudent ON student = sid
target    :P/{ssn} :enrolledIn :C/{course} .
```

Mapping pattern: Clustering Entity to Class (MpCE2C) – Equality case

Relational schema and constraints:
$T_E(\underline{\mathbf{K}}, \mathbf{A})$,
Attributes $\mathbf{B} \subseteq \mathbf{K} \cup \mathbf{A}$
partition table T_E into sub-tables $T_{E_\mathbf{v}}$
such that $t \in T_{E_\mathbf{v}}$ iff $t[\mathbf{B}] = \mathbf{v}$

$\{ V_{E_\mathbf{v}}(\mathbf{K}, \mathbf{A}) = \sigma_{\mathbf{B}=\mathbf{v}}(T_E) \}_{\mathbf{v} \in \pi_\mathbf{B}(T_E)}$



Attributes $\mathbf{B} \subseteq \mathbf{K} \cup \mathbf{A}$
partition entity E into sub-entities $E_\mathbf{v}$
such that $o \in E_\mathbf{v}$ iff $\mathbf{B}(o) = \mathbf{v}$

Mapping assertions:

$\{ s : \sigma_{\mathbf{B}=\mathbf{v}} T_E$
$t :$:E/{K} rdf:type $C_E^\mathbf{v} . \}_{\mathbf{v} \in \pi_\mathbf{B}(T_E)}$

Ontology axioms:

$\{ C_E^\mathbf{v} \sqsubseteq C_E \}_{\mathbf{v} \in \pi_\mathbf{B}(T_E)}$

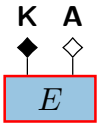
For the application of the mapping pattern, we observe the following:

- This pattern is characterized by a table T_E corresponding to a class C_E, and a derivation rule defining sub-classes of C_E according to the values for attributes $\mathbf{B}$ in T_E.
- Accordingly, instances in T_E can be mapped to ontology objects in the sub-classes $C_E^\mathbf{v}$ of C_E.
- As for other patterns, this pattern produces views according to the possible values $\mathbf{v}$ of $\mathbf{B}$.

Mapping pattern: Clustering Entity to Class (MpCE2C) – Equality case

Relational schema and constraints:
$T_E(\underline{\mathbf{K}}, \mathbf{A})$,
Attributes $\mathbf{B} \subseteq \mathbf{K} \cup \mathbf{A}$
partition table T_E into sub-tables $T_{E_\mathbf{v}}$
such that $t \in T_{E_\mathbf{v}}$ iff $t[\mathbf{B}] = \mathbf{v}$

$$\{\ V_{E_\mathbf{v}}(\mathbf{K}, \mathbf{A}) = \sigma_{\mathbf{B}=\mathbf{v}}(T_E)\ \}_{\mathbf{v} \in \pi_\mathbf{B}(T_E)}$$



Attributes $\mathbf{B} \subseteq \mathbf{K} \cup \mathbf{A}$
partition entity E into sub-entities $E_\mathbf{v}$
such that $o \in E_\mathbf{v}$ iff $\mathbf{B}(o) = \mathbf{v}$

Mapping assertions:

$$\{\ s : \sigma_{\mathbf{B}=\mathbf{v}} T_E$$
$$t : \texttt{:E/\{K\} rdf:type } C_E^\mathbf{v}\ .\ \}_{\mathbf{v} \in \pi_\mathbf{B}(T_E)}$$

Ontology axioms:

$$\{\ C_E^\mathbf{v} \sqsubseteq C_E\ \}_{\mathbf{v} \in \pi_\mathbf{B}(T_E)}$$

For the application of the mapping pattern, we observe the following:

- This pattern is characterized by a table T_E corresponding to a class C_E, and a derivation rule defining sub-classes of C_E according to the values for attributes **B** in T_E.
- Accordingly, instances in T_E can be mapped to ontology objects in the sub-classes $C_E^\mathbf{v}$ of C_E.
- As for other patterns, this pattern produces views according to the possible values **v** of **B**.

Mapping pattern: Clustering Entity to Class (MpCE2C) – Example

Consider a table TStudent containing students with an attribute degree defining whether they are enrolled in a BSc or MSc study course and ranging over 'B' or 'M'.

```
TStudent(sid,name,dob,degree)
```

Mapping: The ontology defines a class Student with two subclasses BScStudent and MScStudent. Pattern **MpCE2C** clusters the table according to the degree attribute, and instantiates the classes BScStudent and MScStudent accordingly.

```
mappingId MStudent
source    SELECT sid, name, dob FROM TStudent
target    :S/{sid} rdf:type :Student ;    :name {name} ;    :dob {dateOfBirth} .

mappingId MBSc
source    SELECT sid FROM TStudent WHERE degree = 'B'
target    :S/{sid} rdf:type :BScStudent .

mappingId MMSc
source    SELECT sid FROM TStudent WHERE degree = 'M'
target    :S/{sid} rdf:type :MScStudent .
```

Mapping pattern: Clustering Entity to Class (MpCE2C) – Example

Consider a table TStudent containing students with an attribute degree defining whether they are enrolled in a BSc or MSc study course and ranging over 'B' or 'M'.

```
TStudent(sid,name,dob,degree)
```

Mapping: The ontology defines a class Student with two subclasses BScStudent and MScStudent. Pattern **MpCE2C** clusters the table according to the degree attribute, and instantiates the classes BScStudent and MScStudent accordingly.

```
mappingId MStudent
source    SELECT sid, name, dob FROM TStudent
target    :S/{sid} rdf:type :Student ;    :name {name} ;    :dob {dateOfBirth} .

mappingId MBSc
source    SELECT sid FROM TStudent WHERE degree = 'B'
target    :S/{sid} rdf:type :BScStudent .

mappingId MMSc
source    SELECT sid FROM TStudent WHERE degree = 'M'
target    :S/{sid} rdf:type :MScStudent .
```

Further mapping patterns

- Similarly to the previous pattern, which clusters instances of a class into different subclasses, we can consider patterns that generate a cluster of data properties, or a cluster of object properties, according to different criteria that can be applied to the source data.
- In order to understand when such patterns can be applied, and then define the corresponding mapping assertions and the expected underlying ontology axioms, we can proceed in a way similar to the case of a cluster of (sub)classes.
- More in general, we might conceive also additional patterns that involve more complex operations or queries over the data.
- Also, in any (sufficiently complex) real-world integration scenario, many cases will occur for which none of the specified pattern applies.
- Therefore, based on (the knowledge that the designer has about) the domain semantics, and the constructs that are available in the ontology, in general also ad-hoc mappings need to be defined.

Additional considerations on IRI-templates

- As we have seen, it is a good practice to include in the IRI-template a prefix that depends on the kind of object (i.e., the class).
- In the case of ISA hierarchies, one has to pay attention on whether to use the same or different templates for the various classes in the hierarchy:
  - Using the same template allows for specifying joins across the various classes of the hierarchy.
  - Using different templates allows for differentiating the different classes and for applying stricter pruning of queries, which helps in query optimization.
- One has also to consider whether to include info about the data source as part of the IRI-template or not:
  - In general, this is not done, which makes the data sources transparent to the user who queries.
  - By including the data source in the IRI-template, such information is recorded in the created objects.

Outline

Design scenarios for VKG mapping patterns

Depending on what information is available, we can consider different design scenarios where the patterns can be applied:

1. **Debugging of a VKG specification** that is already in place.
2. **Conceptual schema reverse engineering** for a DB that represents the domain of interest by using a given full VKG specification.
3. **Mapping bootstrapping** for a given DB and ontology that miss the mappings relating them.
4. **Ontology + mapping bootstrapping** from a given DB with constraints, and possibly a conceptual schema.
5. **VKG bootstrapping**, where the goal is to set up a full VKG specification from a conceptual schema of the domain.

Automating the mapping design process

- In a complex real-world scenario, understanding the domain semantics, the semantics of the data sources, and how the sources have to be related to the global schema/ontology can be rather resource intensive and therefore costly.
- Currently, there are no tools that completely automate this process, and it is unlikely that a completely automated solution is possible at all.
- However, there are tools that provide automated support for the (already difficult) task of understanding which elements in one schema (e.g., a source) can correspond to which elements of another schema (e.g., the global schema). This task is called **schema matching**.
- Based on a proposed match between elements, mapping patterns can provide valuable indications on how to convert the match into an actual mapping, i.e., how to define the (SQL) queries that correctly relate the semantics of the sources to that of the ontology.
- Also, mapping patterns can be automatically discovered, either by considering the constraints on the data sources, or, more interestingly, derive the constraints from the actual data, even when they are not defined over the sources at the schema level.
- Work in this direction is ongoing.

Outline

Summary

- VKGs are by now a mature technology to address the challenges in data access and integration.
- They rely on W3C standards and on supporting APIs and libraries.
- The technology is general purpose and applied in many different scenarios, but it can be tailored towards specific domains by relying on standard ontologies.
- Performance and scalability w.r.t. larger datasets and larger and more complex ontologies, is still a key challenge that is addressed by various kinds of optimizations in the query processing engine.
- The design of VKG-based solutions, notably the mappings, is a major bottleneck that requires a principled approach and supporting methodologies $\rightsquigarrow$ Mapping patterns

Summary

- VKGs are by now a mature technology to address the challenges in data access and integration.
- They rely on W3C standards and on supporting APIs and libraries.
- The technology is general purpose and applied in many different scenarios, but it can be tailored towards specific domains by relying on standard ontologies.
- Performance and scalability w.r.t. larger datasets and larger and more complex ontologies, is still a key challenge that is addressed by various kinds of optimizations in the query processing engine.
- The design of VKG-based solutions, notably the mappings, is a major bottleneck that requires a principled approach and supporting methodologies $\rightsquigarrow$ Mapping patterns

Ongoing and future work

- Accessing alternative types of data:
  - temporal data [C., Okulmus, et al. 2023]
  - noSQL, tree, and graph structured data [Botoeva et al. 2019]
  - raster data and geo-spatial data [PhD by Arka Ghosh]
- Ontology-based federation, for accessing multiple, heterogeneous data sources [Gu et al. 2022]
- Privacy issues [Cima et al. 2020; Bonatti et al. 2022], [PhD by Divya Baura]
- Ontology-based update [PhD by Romuald Wandji]
- (Semi-)automatic extraction/learning of ontology axioms and mappings [Calvanese et al. 2021]

Ongoing and future work

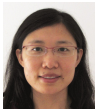
- Accessing alternative types of data:
  - temporal data [C., Okulmus, et al. 2023]
  - noSQL, tree, and graph structured data [Botoeva et al. 2019]
  - raster data and geo-spatial data [PhD by Arka Ghosh]
- Ontology-based federation, for accessing multiple, heterogeneous data sources [Gu et al. 2022]
- Privacy issues [Cima et al. 2020; Bonatti et al. 2022], [PhD by Divya Baura]
- Ontology-based update [PhD by Romuald Wandji]
- (Semi-)automatic extraction/learning of ontology axioms and mappings [Calvanese et al. 2021]
- For complex real-world scenarios, VKG-design requires also tool support.

  Hands-on part of this tutorial with ONTOPIC Studio.

A great thank you to our collaborators


Elena Botoeva


Julien Corman


Linfang Ding


Zhenzhen Gu


Elem Güzel


Marco Montali


Alessandro Mosca


Mariano Rodriguez Muro


Guohui Xiao

Technion Haifa


Avigdor Gal


Roee Shraga

Birkbeck College London


Roman Kontchakov


Vladislav Ryzhikov


Michael Zakharyaschev

Ontopic s.r.l.


Benjamin Cogrel


Sarah Komla Ebri

U. Roma “La Sapienza”


Giuseppe De Giacomo


Domenico Lembo


Maurizio Lenzerini


Antonella Poggi


Riccardo Rosati

Acknowledgements

CyclOps: Automated End-to-End Data Life Cycle Management for FAIR Data Integration, Processing, and Re-Use
Grant agreement ID: 101135513
Funded by the EU under the Horizon Europe programme.

Project Highlights:

- Handles large volumes of data from diverse sources across the complete data lifecycle.
- Uses AI and Knowledge Graphs to automate generation and execution of data processing pipelines.
- Incorporates human input to ensure data governance and quality.
- Enables organisations to share, access, and analyse machine- and human-generated data seamlessly.
- Facilitates development of new data-centric business models and value-added services.

CyclOps

Funded by the European Union

VKGs underpin CyclOps, offering a unified RDF view over diverse sources for seamless integration, processing, analysis, and FAIR-compliant data re-use.

Time for Hands-On Session with

- *Ontop* website: `https://ontop-vkg.org/`
- Github: `https://github.com/ontop/ontop/`
- Cyclops website: `https://www.cyclopsproject.eu/`
- *Ontopic* website: `https://ontopic.ai/`

diego.calvanese@unibz.it
davide.lanti@unibz.it
benjamin.cogrel@ontopic.ai

References I

[1] Guohui Xiao, Diego C., Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati & Michael Zakharyaschev. “Ontology-Based Data Access: A Survey”. In: *Proc. of the 27th Int. Joint Conf. on Artificial Intelligence (IJCAI)*. IJCAI Org., 2018, pp. 5511–5519. DOI: 10.24963/ijcai.2018/777.

[2] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue & Carsten Lutz. *OWL 2 Web Ontology Language Profiles (Second Edition)*. W3C Recommendation. Available at http://www.w3.org/TR/owl2-profiles/. World Wide Web Consortium, Dec. 2012.

[3] Franz Baader, Diego C., Deborah McGuinness, Daniele Nardi & Peter F. Patel-Schneider, eds. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[4] Maurizio Lenzerini & Paolo Nobili. “On the Satisfiability of Dependency Constraints in Entity-Relationship Schemata”. In: *Information Systems* 15.4 (1990), pp. 453–461.

[5] Sonia Bergamaschi & Claudio Sartori. “On Taxonomic Reasoning in Conceptual Design”. In: *ACM Trans. on Database Systems* 17.3 (1992), pp. 385–422.

References II

[6] Alexander Borgida. “Description Logics in Data Management”. In: *IEEE Trans. on Knowledge and Data Engineering* 7.5 (1995), pp. 671–682.

[7] Diego C., Maurizio Lenzerini & Daniele Nardi. “Unifying Class-Based Representation Formalisms”. In: *J. of Artificial Intelligence Research* 11 (1999), pp. 199–240.

[8] Alexander Borgida & Ronald J. Brachman. “Conceptual Modeling with Description Logics”. In: *The Description Logic Handbook: Theory, Implementation and Applications*. Ed. by Franz Baader, Diego C., Deborah McGuinness, Daniele Nardi & Peter F. Patel-Schneider. Cambridge University Press, 2003. Chap. 10, pp. 349–372.

[9] Daniela Berardi, Diego C. & Giuseppe De Giacomo. “Reasoning on UML Class Diagrams”. In: *Artificial Intelligence* 168.1–2 (2005), pp. 70–118.

[10] Anna Queralt, Alessandro Artale, Diego C. & Ernest Teniente. “OCL-Lite: Finite Reasoning on UML/OCL Conceptual Schemas”. In: *Data and Knowledge Engineering* 73 (2012), pp. 1–22.

[11] Diego C., Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini & Riccardo Rosati. “Tractable Reasoning and Efficient Query Answering in Description Logics: The *DL-Lite* Family”. In: *J. of Automated Reasoning* 39.3 (2007), pp. 385–429.

References III

[12] Antonella Poggi, Domenico Lembo, Diego C., Giuseppe De Giacomo, Maurizio Lenzerini & Riccardo Rosati. “Linking Data to Ontologies”. In: *J. on Data Semantics* 10 (2008), pp. 133–173. DOI: 10.1007/978-3-540-77688-8_5.

[13] Nicola Guarino & Christopher A. Welty. “An Overview of OntoClean”. In: *Handbook on Ontologies*. Ed. by Steffen Staab & Rudi Studer. International Handbooks on Information Systems. Springer, 2009, pp. 201–220. DOI: 10.1007/978-3-540-92673-3_9.

[14] Maurizio Lenzerini. “Data Integration: A Theoretical Perspective.”. In: *Proc. of the 21st ACM Symp. on Principles of Database Systems (PODS)*. 2002, pp. 233–246. DOI: 10.1145/543613.543644.

[15] Erhard Rahm & Philip A. Bernstein. “A Survey of Approaches to Automatic Schema Matching”. In: *Very Large Database J.* 10.4 (2001), pp. 334–350.

[16] Jérôme Euzenat & Pavel Shvaiko. *Ontology Matching*. Springer, 2007.

[17] Dimitrios-Emmanuel Spanos, Periklis Stavrou & Nikolas Mitrou. “Bringing Relational Databases into the Semantic Web: A Survey”. In: *Semantic Web J.* 3.2 (2012), pp. 169–209.

References IV

[18] Marcelo Arenas, Alexandre Bertails, Eric Prud'hommeaux & Juan Sequeda. *A Direct Mapping of Relational Data to RDF*. W3C Recommendation. Available at http://www.w3.org/TR/rdb-direct-mapping/. World Wide Web Consortium, Sept. 2012.

[19] Diego C., Cem Okulmus, Magdalena Ortiz & Mantas Simkus. "On the Way to Temporal OBDA Systems". In: *Proc. of the 15th Alberto Mendelzon Int. Workshop on Foundations of Data Management (AMW)*. Vol. 3409. CEUR Workshop Proceedings. CEUR-WS.org, 2023.

[20] Elena Botoeva, Diego C., Benjamin Cogrel, Julien Corman & Guohui Xiao. "Ontology-based Data Access – Beyond Relational Sources". In: *Intelligenza Artificiale* 13.1 (2019), pp. 21–36. DOI: 10.3233/IA-190023.

[21] Zhenzhen Gu, Davide Lanti, Alessandro Mosca, Guohui Xiao, Jing Xiong & Diego Calvanese. "Ontology-based Data Federation". In: *Proc. of the 11th Int. Joint Conf. on Knowledge Graphs (IJCKG)*. ACM, 2022, pp. 10–19. DOI: 10.1145/3579051.3579070.

References V

[22] Gianluca Cima, Domenico Lembo, Lorenzo Marconi, Riccardo Rosati & Domenico Fabio Savo. “Controlled Query Evaluation in Ontology-Based Data Access”. In: *Proc. of the 19th Int. Semantic Web Conf. (ISWC)*. Vol. 12506. Lecture Notes in Computer Science. Springer, 2020, pp. 128–146. DOI: 10.1007/978-3-030-62419-4_8.

[23] Piero A. Bonatti, Gianluca Cima, Domenico Lembo, Lorenzo Marconi, Riccardo Rosati, Luigi Sauro & Domenico Fabio Savo. “Controlled Query Evaluation in OWL 2 QL: A “Longest Honeymoon” Approach”. In: *Proc. of the 21st Int. Semantic Web Conf. (ISWC)*. Vol. 13489. Lecture Notes in Computer Science. Springer, 2022, pp. 428–444. DOI: 10.1007/978-3-031-19433-7_25.

[24] Diego Calvanese, Avigdor Gal, Naor Haba, Davide Lanti, Marco Montali, Alessandro Mosca & Roee Shraga. “ADAMAP: Automatic Alignment of Relational Data Sources using Mapping Patterns”. In: *Proc. of the 33rd Int. Conf. on Advanced Information Systems Engineering (CAiSE)*. Vol. 12751. Lecture Notes in Computer Science. Springer, 2021, pp. 193–209. DOI: 10.1007/978-3-030-79382-1_12.