Ontology-based Data Access Made Practical

Diego Calvanese

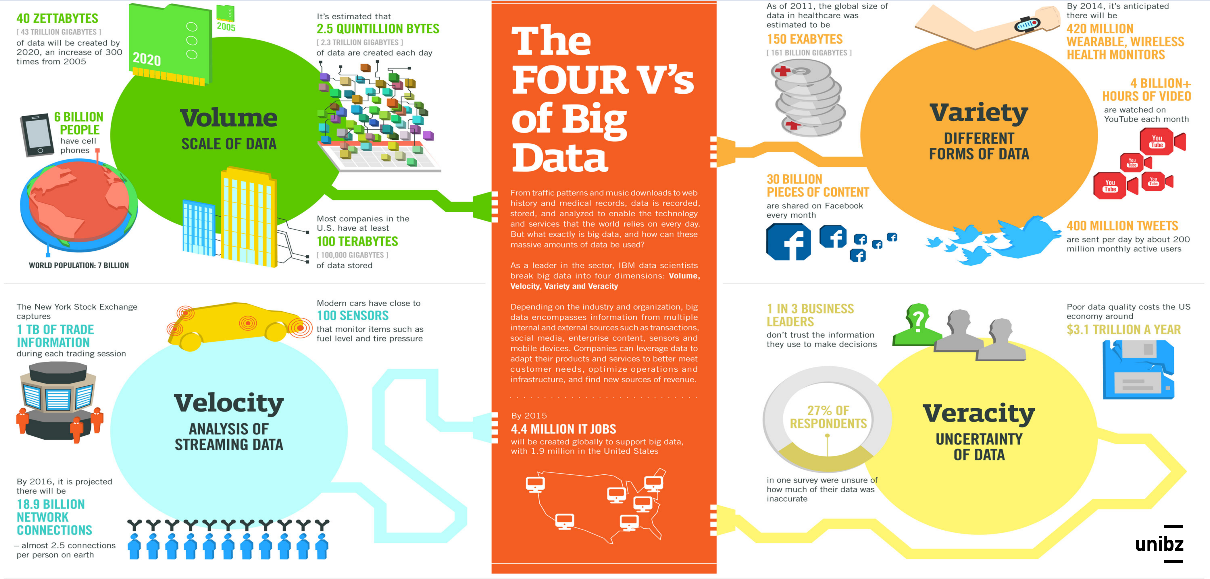
KRDB Research Centre for Knowledge and Data
Free University of Bozen-Bolzano, Italy

Department of Computing Science
Umeå University, Sweden

Ontopic s.r.l.

EDBT-INTENDED Summer School
4–9 July 2022 – Bordeaux (France)

Challenges in data management



The FOUR V's of Big Data

From traffic patterns and music downloads to web history and medical records, data is recorded, stored, and analyzed to enable the technology and services that the world relies on every day. But what exactly is big data, and how can these massive amounts of data be used?

As a leader in the sector, IBM data scientists break big data into four dimensions: **Volume, Velocity, Variety and Veracity**

Depending on the industry and organization, big data encompasses information from multiple internal and external sources such as transactions, social media, enterprise content, sensors and mobile devices. Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.

By 2015
4.4 MILLION IT JOBS
will be created globally to support big data, with 1.9 million in the United States

Volume
SCALE OF DATA

40 ZETTABYTES
[43 TRILLION GIGABYTES]
of data will be created by 2020, an increase of 300 times from 2005

It's estimated that
2.5 QUINTILLION BYTES
[2.3 TRILLION GIGABYTES]
of data are created each day

6 BILLION PEOPLE
have cell phones

WORLD POPULATION: 7 BILLION

Most companies in the U.S. have at least
100 TERABYTES
[100,000 GIGABYTES]
of data stored

Velocity
ANALYSIS OF STREAMING DATA

The New York Stock Exchange captures
1 TB OF TRADE INFORMATION
during each trading session

Modern cars have close to
100 SENSORS
that monitor items such as fuel level and tire pressure

By 2016, it is projected there will be
18.9 BILLION NETWORK CONNECTIONS
– almost 2.5 connections per person on earth

Variety
DIFFERENT FORMS OF DATA

As of 2011, the global size of data in healthcare was estimated to be
150 EXABYTES
[161 BILLION GIGABYTES]

By 2014, it's anticipated there will be
420 MILLION WEARABLE, WIRELESS HEALTH MONITORS

4 BILLION+ HOURS OF VIDEO
are watched on YouTube each month

30 BILLION PIECES OF CONTENT
are shared on Facebook every month

400 MILLION TWEETS
are sent per day by about 200 million monthly active users

Veracity
UNCERTAINTY OF DATA

1 IN 3 BUSINESS LEADERS
don't trust the information they use to make decisions

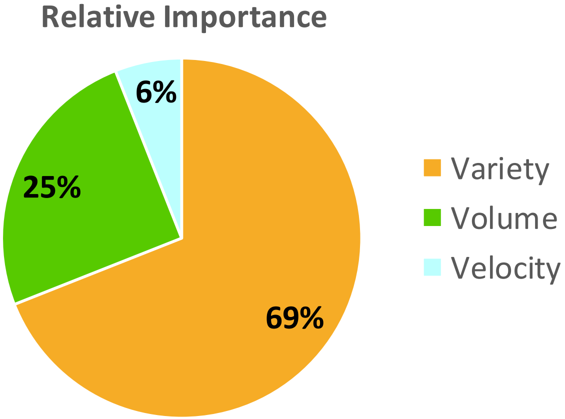
Poor data quality costs the US economy around
$3.1 TRILLION A YEAR

27% OF RESPONDENTS
in one survey were unsure of how much of their data was inaccurate

Sources: McKinsey Global Institute, Twitter, Cisco, Gartner, EMC, SAS, IBM, MEPTEC, QAS

Variety, not volume, is driving data management initiatives

MIT Sloan Management Review (28 March 2016)



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

How much time is spent searching for the right data?

Important problem: searching for data and establishing its quality

Example: in oil&gas, engineers spend 30–70% of their time on this problem (Crompton, 2008)

Challenge: Accessing legacy data sources

Statoil (now Equinor) Exploration

Geologists at Statoil, prior to making decisions on drilling new wellbores, need to gather relevant information about previous drillings.

Slegge relational database:

- Terabytes of relational data
- 1,545 tables and 1727 views
- each with dozens of attributes
- consulted by 900 geologists

Problem: Translating information needs

Information need expressed by geologists

In my geographical area of interest, return all pressure data tagged with key stratigraphy information with understandable quality control attributes, and suitable for further filtering.

To obtain the answer, this needs to be translated into SQL:

- Main table for wellbores has 38 columns (with cryptic names).
- To obtain pressure data requires a 4-table join with two additional filters.
- To obtain stratigraphic information requires a join with 5 more tables.

Problem: Translating information needs

We would obtain the following SQL query:

```
SELECT WELLBORE.IDENTIFIER, PTY_PRESSURE.PTY_PRESSURE_S,
       STRATIGRAPHIC_ZONE.STRAT_COLUMN_IDENTIFIER, STRATIGRAPHIC_ZONE.STRAT_UNIT_IDENTIFIER
FROM WELLBORE,
     PTY_PRESSURE,
     ACTIVITY FP_DEPTH_DATA
        LEFT JOIN (PTY_LOCATION_1D FP_DEPTH_PT1_LOC
           INNER JOIN PICKED_STRATIGRAPHIC_ZONES ZS
              ON ZS.STRAT_ZONE_ENTRY_MD <= FP_DEPTH_PT1_LOC.DATA_VALUE_1_O AND
                 ZS.STRAT_ZONE_EXIT_MD >= FP_DEPTH_PT1_LOC.DATA_VALUE_1_O AND
                 ZS.STRAT_ZONE_DEPTH_UOM = FP_DEPTH_PT1_LOC.DATA_VALUE_1_OU
           INNER JOIN STRATIGRAPHIC_ZONE
              ON   ZS.WELLBORE = STRATIGRAPHIC_ZONE.WELLBORE AND
                 ZS.STRAT_COLUMN_IDENTIFIER = STRATIGRAPHIC_ZONE.STRAT_COLUMN_IDENTIFIER AND
                 ZS.STRAT_INTERP_VERSION = STRATIGRAPHIC_ZONE.STRAT_INTERP_VERSION   AND
                 ZS.STRAT_ZONE_IDENTIFIER = STRATIGRAPHIC_ZONE.STRAT_ZONE_IDENTIFIER)
           ON FP_DEPTH_DATA.FACILITY_S = ZS.WELLBORE AND
              FP_DEPTH_DATA.ACTIVITY_S  = FP_DEPTH_PT1_LOC.ACTIVITY_S,
     ACTIVITY_CLASS FORM_PRESSURE_CLASS
WHERE WELLBORE.WELLBORE_S = FP_DEPTH_DATA.FACILITY_S AND
      FP_DEPTH_DATA.ACTIVITY_S = PTY_PRESSURE.ACTIVITY_S AND
      FP_DEPTH_DATA.KIND_S = FORM_PRESSURE_CLASS.ACTIVITY_CLASS_S AND
      WELLBORE.REF_EXISTENCE_KIND = 'actual' AND
      FORM_PRESSURE_CLASS.NAME = 'formation pressure depth data'
```

Problem: Translating information needs

We would obtain the following SQL query:

```
SELECT WELL
       STRA
FROM WELLBO
     PTY_PR
     ACTIVI
        LEF

           INNER JOIN STRATIGRAPHIC_ZONE
              ON   ZS.WELLBORE = STRATIGRAPHIC_ZONE.WELLBORE AND

     ACTIVI
WHERE WELLB
      FP_DE
      FP_DEPTH_DATA.KIND_S = FORM_PRESSURE_CLASS.ACTIVITY_CLASS_S AND
      WELLBORE.REF_EXISTENCE_KIND = 'actual' AND
      FORM_PRESSURE_CLASS.NAME = 'formation pressure depth data'
```

This can be very time consuming, and requires knowledge of the domain of interest, a deep understanding of the database structure, and general IT expertise.

This is also very costly!

Equinor loses **50.000.000€** per year only due to this problem!!

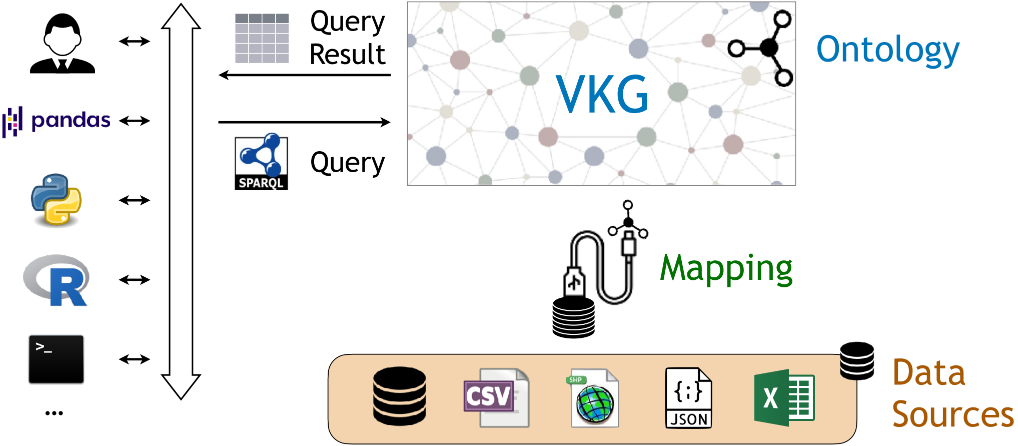
How to address data access (and integration) challenges?

We combine three key ideas:

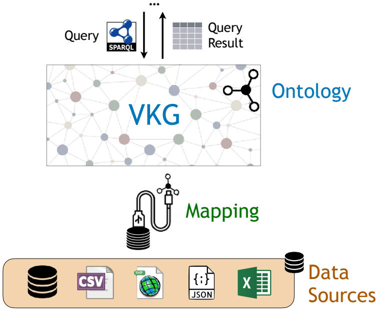
1. Use a global (or integrated) schema and **map the data sources to the global schema**.
2. Adopt a very flexible data model for the global schema
   $\leadsto$ **Knowledge Graph** whose vocabulary is expressed in an **ontology**.
3. Exploit **virtualization**, i.e., the KG is not materialized, but 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, C., et al. 2018, IJCAI]

Virtual Knowledge Graph (VKG) architecture



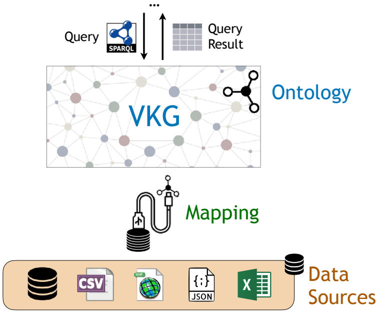
Why an ontology?



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 a Knowledge Graph for the global schema?

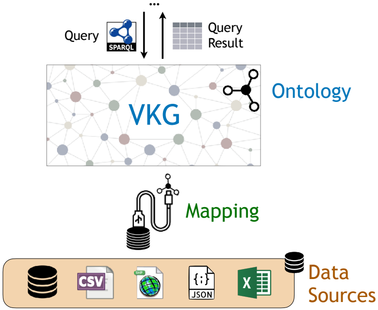


The traditional approach to data integration adopts a relational global schema.

A Knowledge Graph, instead:

- Does not require to commit early on to a specific structure.
- Can better accommodate heterogeneity.
- Can better deal with missing / incomplete information.
- Does not require complex restructuring operations to accommodate new information or new data sources.

Why mappings?

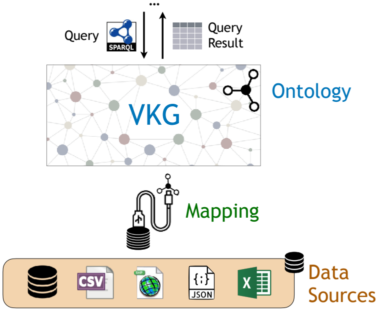


The traditional approach to data integration relies on mediators, which are specified through complex code.

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.

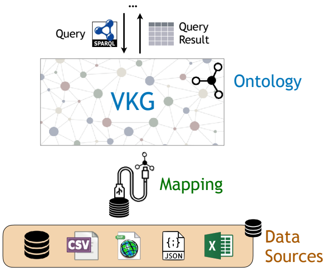
In the virtual approach, instead:

- 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.

Outline

Outline

Incomplete information



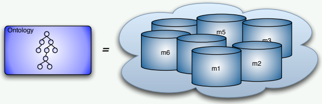
We are in a setting of **incomplete information**!!!

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!



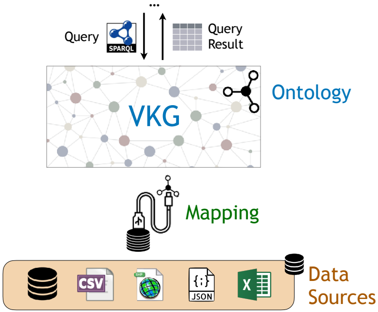
Minus:

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)

Outline

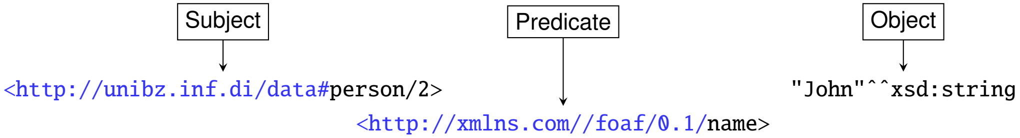
Resource Description Framework (RDF)

- 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**:



Subject

`<http://unibz.inf.di/data#person/2>`

Predicate

`<http://xmlns.com//foaf/0.1/name>`

Object

`"John"^^xsd:string`

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

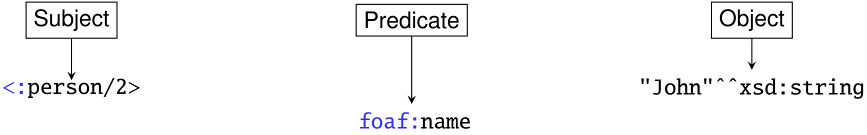
1. Subject: IRI of the described resource
2. Predicate: IRI of the property
3. Object: attribute value or IRI of another resource

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**:



Subject → `<:person/2>`

Predicate → `foaf:name`

Object → `"John"^^xsd:string`

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

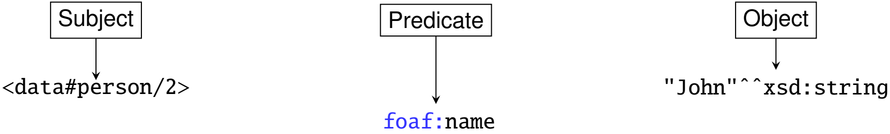
1. Subject: IRI of the described resource
2. Predicate: IRI of the property
3. Object: attribute value or IRI of another resource

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**:



Subject

`<data#person/2>`

Predicate

`foaf:name`

Object

`"John"^^xsd:string`

Triple elements: resources denoted by **global identifiers** (called **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")

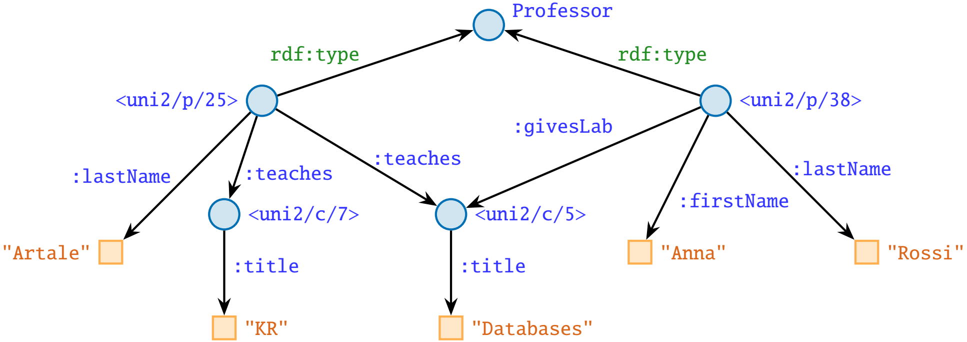
Object property of an individual:

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

RDF 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:



Additional RDF features

RDF has additional features that we do not cover here:

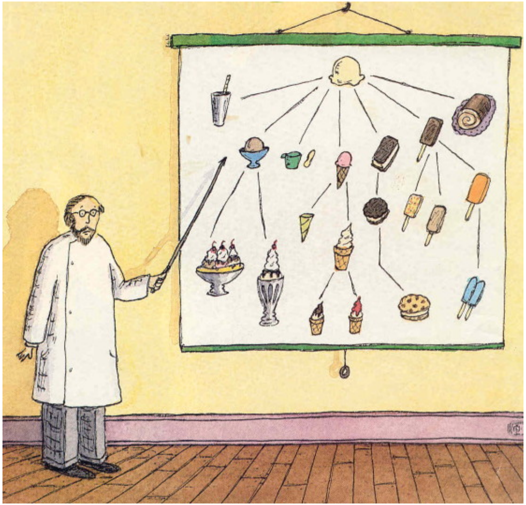
- datatypes
- blank nodes
- named graphs

Outline

unibz

What is an ontology?

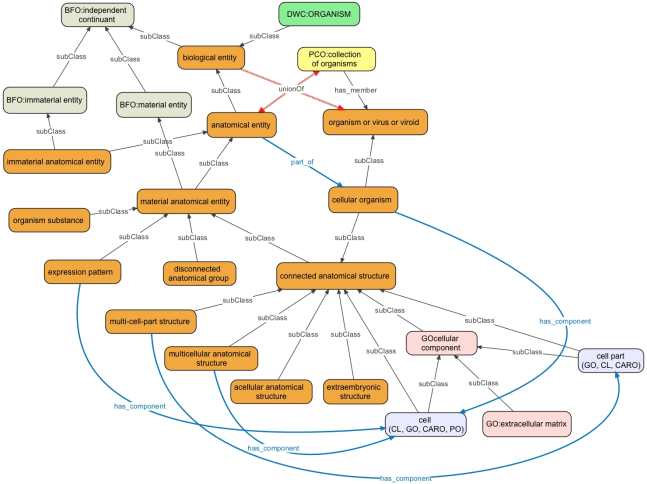
- An ontology conceptualizes a domain of interest in terms of **concepts/classes**, (binary) **relations**, and their **properties**.
- It typically organizes the concepts in a hierarchical structure.
- Ontologies are often represented as graphs.
- However, an ontology is actually a **logical theory**,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



unibz

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**.

$$\begin{array}{rcl}
\mathsf{Actor} & \sqsubseteq & \mathsf{Staff} \\
\mathsf{SeriesActor} & \sqsubseteq & \mathsf{Actor} \\
\mathsf{MovieActor} & \sqsubseteq & \mathsf{Actor} \\
\mathsf{SeriesActor} & \sqsubseteq & \neg\mathsf{MovieActor}
\end{array}$$

$$\begin{array}{rcl}
\mathsf{Staff} & \sqsubseteq & \exists \mathit{ssn} \\
\exists \mathit{ssn}^- & \sqsubseteq & \texttt{xsd:int} \\
\multicolumn{3}{c}{(\mathbf{funct}\ \mathit{ssn})}
\end{array}$$

$$\begin{array}{rcl}
\exists \mathsf{actsIn} & \sqsubseteq & \mathsf{MovieActor} \\
\exists \mathsf{actsIn}^- & \sqsubseteq & \mathsf{Movie} \\
\mathsf{MovieActor} & \sqsubseteq & \exists \mathsf{actsIn} \\
\mathsf{Movie} & \sqsubseteq & \exists \mathsf{actsIn}^- \\
\mathsf{actsIn} & \sqsubseteq & \mathsf{playsIn} \\
 & \ldots &
\end{array}$$

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) of the *DL-Lite*-family [C., De Giacomo, et al. 2007].
- 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 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.

OWL 2 QL knowledge bases

An **OWL 2 QL knowledge base (KB)** 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.

Axioms in an OWL 2 QL ontology

In an OWL 2 QL ontology, one can express knowledge about the classes and properties in the domain of interest by means of the following types of axioms.

Axiom type	**OWL syntax**	**DL syntax**
Class declaration	`:Actor` **`rdf:type`** `owl:Class`	Actor
Object property decl.	`:actsIn` **`rdf:type`** `owl:ObjectProperty`	actsIn
Data property declaration	`:title` **`rdf:type`** `owl:DatatypeProperty`	title
Subclass assertion	`:MovieActor` **`rdfs:subClassOf`** `:Actor`	MovieActor $\sqsubseteq$ Actor
Class disjointness	`:Actor` **`owl:disjointWith`** `:Movie`	Actor $\sqsubseteq$ $\neg$Movie
Domain of a property	`:actsIn` **`rdfs:domain`** `:MovieActor`	$\exists$actsIn $\sqsubseteq$ MovieActor
Range of a property	`:actsIn` **`rdfs:range`** `:Movie`	$\exists$actsIn$^-$ $\sqsubseteq$ Movie
Mandatory participation	**`owl:someValuesFrom`** in superclass expression	MovieActor $\sqsubseteq$ $\exists$actsIn
Subproperty assertion	`:actsIn` **`rdfs:subPropertyOf`** `:playsIn`	actsIn $\sqsubseteq$ playsIn
Inverse properties	`:actsIn` **`owl:inverseOf`** `:hasActor`	actsIn $\equiv$ hasActor$^-$

Syntax and semantics of OWL 2 QL axioms

Axiom type	**OWL Syntax**	**DL Syntax**	**FOL**
Class declaration	`C rdf:type owl:Class`	C	$C(x)$
Object property declaration	`P rdf:type owl:ObjectProperty`	P	$P(x, y)$
Data property declaration	`A rdf:type owl:DatatypeProperty`	A	$A(x, y)$
Subclass assertion	`C1 rdfs:subClassOf C2`	$C_1 \sqsubseteq C_2$	$\forall x. C_1(x) \rightarrow C_2(x)$
Class disjointness	`C1 owl:disjointWith C2`	$C_1 \sqsubseteq \neg C_2$	$\forall x. C_1(x) \rightarrow \neg C_2(x)$
Domain of a property	`P rdfs:domain C1`	$\exists P \sqsubseteq C_1$	$\forall x. (\exists y. P(x, y)) \rightarrow C_1(x)$
Range of a property	`P rdfs:range C2`	$\exists P^- \sqsubseteq C_2$	$\forall y. (\exists x. P(x, y)) \rightarrow C_2(y)$
Mandatory participation	using `owl:someValuesFrom`	$C \sqsubseteq \exists R$	$\forall x. C(x) \rightarrow \exists y. R(x, y)$
Subproperty assertion	`P1 rdfs:subPropertyOf R2`	$P_1 \sqsubseteq R_2$	$\forall x, y. P_1(x, y) \rightarrow R_2(x, y)$
Inverse property	`P2 owl:inverseOf P1`	$P_1 \equiv P_2^-$	$\forall x, y. P_1(x, y) \leftrightarrow P_2(y, x)$

- 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.

Impact of disjointness and functionalty on query answering

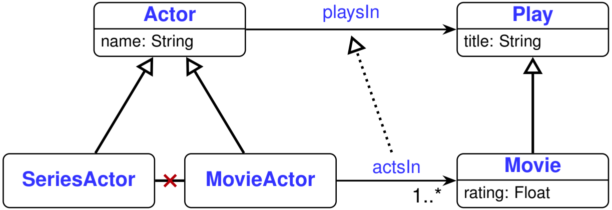
- Disjointness of classes and of properties cannot be expressed in RDFS, but can be expressed in OWL 2 QL.
- Functionality of properties cannot be expressed in OWL 2 QL, but can be expressed in OWL 2 (which is a much more powerful ontology language).
- However, both disjointness and functionality are supported by VKG systems such as *Ontop*.
- These constructs have an impact on consistency, i.e., they might be violated by the data and thus lead to an RDF graph that is inconsistent with the ontology.
- It turns out, however, that neither disjointness nor functionality affect query answering, as long as the ontology and the data are consistent. This means that they are actually ignored by the query evaluation algorithm.

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].

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

$\dots$



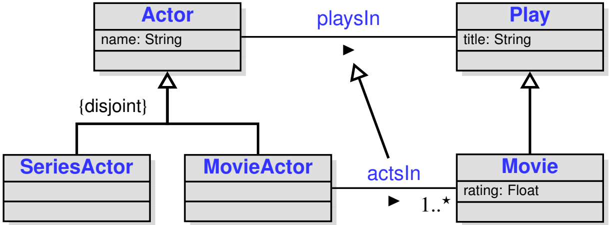
In fact, to visualize an OWL 2 QL ontology, we could have used standard UML class diagrams, instead of the specific graphical notation that we have introduced.

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].

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

$\dots$

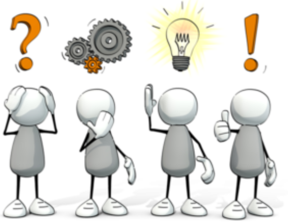


In fact, to visualize an OWL 2 QL ontology, we could have used standard UML class diagrams, instead of the specific graphical notation that we have introduced.

Outline

1. Challenges in Data Access
2. Virtual Knowledge Graphs for Data Access (and Integration)
    - Representing Data in RDF and RDFS
    - Representing Ontologies in OWL 2 QL
    - Query Language – SPARQL
    - Mapping an Ontology to a Relational Database
    - Formalizing the VKG Framework
3. Optimizing Query Answering in VKGs
4. Designing VKG Mappings
5. The Ontop System
6. Conclusions

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 ontologies:

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).

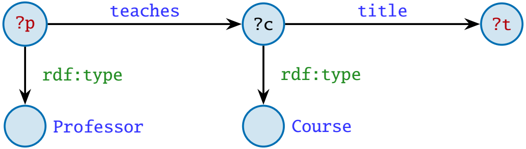
Conjunctive queries – Are concretely represented in SPARQL

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)
- ...

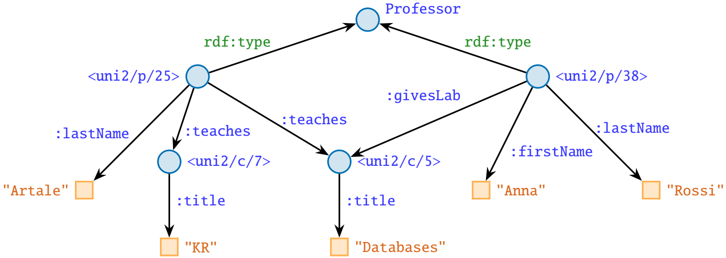
SPARQL Basic Graph Patterns

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).

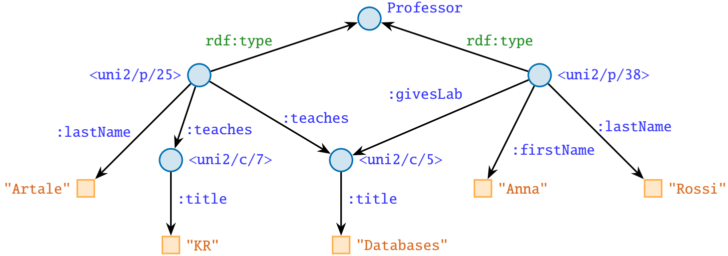
Projecting out variables in a SPARQL query

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"

BGPs vs. conjunctive queries

We can write queries using the more compact and abstract syntax of conjunctive queries (CQs).

Example: BGP

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

vs. conjunctive query

$$\begin{aligned} \boldsymbol{q}(p, ln, c, t) \leftarrow\ & \text{lastName}(p, ln), \\ & \text{teaches}(p, c), \\ & \text{title}(c, t) \end{aligned}$$

A **conjunctive query** $\boldsymbol{q}$ has the form $\boldsymbol{q}(\vec{x}) \leftarrow p_1(\vec{y}_1), \ldots, p(\vec{y}_k)$ where

- $\boldsymbol{q}(\vec{x})$ is called the head of $\boldsymbol{q}$,
- $p_1(\vec{y}_1), \ldots, p(\vec{y}_k)$ is a conjunction of atoms called the body of $\boldsymbol{q}$,
- all variables $\vec{x}$ in the head are among $\vec{y}_1, \ldots, \vec{y}_k$, and
- the variables in $\vec{y}_1, \ldots, \vec{y}_k$ that are not among $\vec{x}$ are existentially quantified.

BGPs vs. conjunctive queries (cont.)

Example: BGP with projection

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

vs. conjunctive query with existential variables

$$\begin{aligned} q(ln, t) \leftarrow\ & \text{lastName}(p, ln), \\ & \text{teaches}(p, c), \\ & \text{title}(c, t) \end{aligned}$$

But there is a difference in semantics when we have an ontology:

- In a SPARQL query, all variables, including those that are projected out, must match nodes of the RDF graph.
- In a conjunctive query, the existentially quantified variables can also match nodes that are existentially implied by the axioms of the ontology.

BGPs vs. conjunctive queries – Example

Consider the KB $\langle\mathcal{O},\mathcal{A}\rangle$, where the ontology is $\mathcal{O} = \{\ C \sqsubseteq \exists P\ \}$ and the RDF graph is $\mathcal{A} = \{\ C(a)\ \}$. Consider further the following SPARQL BGP and the corresponding conjunctive query.

SPARQL query that has the form of a BGP

```
SELECT ?x WHERE { ?x rdf:type :C .  ?x :P ?y . }
```

Conjunctive query

$\boldsymbol{q}(x) \leftarrow C(x),\ P(x,y)$

Every model $\mathcal{M}$ of $\langle\mathcal{O},\mathcal{A}\rangle$ contains the fact $C(a)$ (recall that $a^{\mathcal{M}} = a$), and since $C \sqsubseteq \exists P \in \mathcal{O}$ also a fact $P(a,o)$, for some (existentially implied) object o. For example, the following are models of $\langle\mathcal{O},\mathcal{A}\rangle$:

- $\mathcal{M}_1$, with facts $C(a)$ and $P(a,o_1)$;
- $\mathcal{M}_2$, with facts $C(a)$ and $P(a,o_2)$;
- $\mathcal{M}_3$, with facts $C(a)$, $P(a,o_1)$, and $P(a,o_3)$;
- $\mathcal{M}_4$, with facts $C(a)$ and $P(a,a)$;
- $\cdots$
- $\cdots$

Hence, for every model $\mathcal{M}$ of $\langle\mathcal{O},\mathcal{A}\rangle$, there is a homomorphism from the body of the conjunctive query $\boldsymbol{q}$ to $\mathcal{M}$ that maps x to a. (Therefore, we have that $a \in \text{cert}(\boldsymbol{q},\langle\mathcal{O},\mathcal{A}\rangle)$. – See later.)
Instead, even in the presence of an ontology, the SPARQL query must match on the RDF graph $\mathcal{A}$ to produce an answer. Since $\mathcal{A}$ contains only $C(a)$, the answer to the SPARQL query is empty.

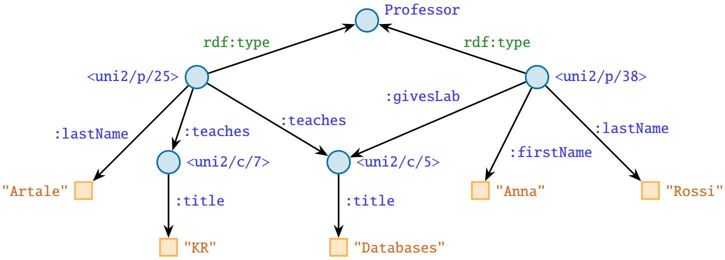
Extending BGPs with OPTIONAL

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"

SPARQL algebra

We have just seen the following features of the SPARQL algebra:

- Basic Graph Patterns
- `OPTIONAL`

The overall algebra has additional features:

- `UNION`
- `ORDER BY`, `LIMIT`, `OFFSET`
- `FILTER` conditions
- `GROUP BY`, to express aggregations and support aggregation operators
- `MINUS`, to remove possible solutions
- path expressions, corresponding to regular expressions

Outline

1. Challenges in Data Access

2. Virtual Knowledge Graphs for Data Access (and Integration)
    - Representing Data in RDF and RDFS
    - Representing Ontologies in OWL 2 QL
    - Query Language – SPARQL
    - Mapping an Ontology to a Relational Database
    - Formalizing the VKG Framework

3. Optimizing Query Answering in VKGs

4. Designing VKG Mappings

5. The Ontop System

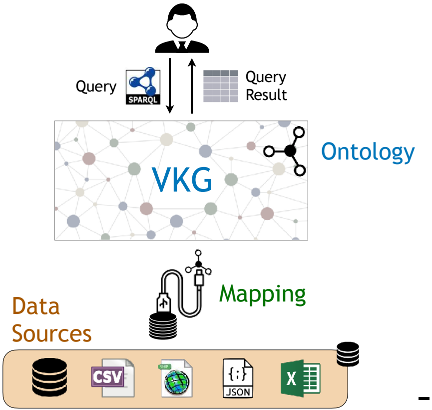
6. Conclusions

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.

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:

$\mathbf{iri}_1(\vec{x}_1)$ $\mathtt{rdf{:}type}$ C where C is a class of the ontology, or
$\mathbf{iri}_1(\vec{x}_1)$ $prop$ $\mathbf{iri}_2(\vec{x}_2)$ where $prop$ 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:

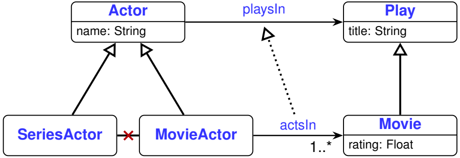
```
SELECT M.mcode, A.acode
FROM MOVIE M, ACTOR A
WHERE M.mcode = A.pcode
  AND M.type = "m"
```

- Target part:

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

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

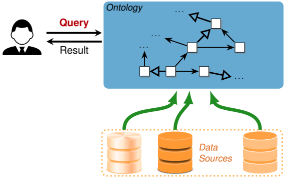
[2] http://sparqlify.org/wiki/Sparqlification_mapping_language

[3] https://github.com/ontop/ontop/wiki/ontopOBDAModel#Mapping_axioms

Outline

1. Challenges in Data Access
2. Virtual Knowledge Graphs for Data Access (and Integration)
   - Representing Data in RDF and RDFS
   - Representing Ontologies in OWL 2 QL
   - Query Language – SPARQL
   - Mapping an Ontology to a Relational Database
   - Formalizing the VKG Framework
3. Optimizing Query Answering in VKGs
4. Designing VKG Mappings
5. The Ontop System
6. Conclusions

VKGs: Formalization



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

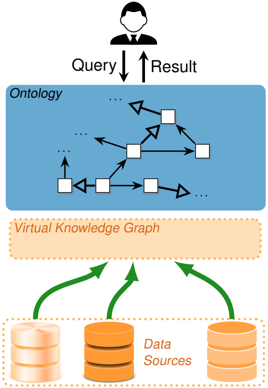
A **VKG specification** is a triple $\mathcal{P} = \langle \mathcal{O}, \mathcal{M}, \mathcal{S} \rangle$, where:

- $\mathcal{O}$ 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 $\langle \mathcal{P}, \mathcal{D} \rangle$, where

- $\mathcal{P} = \langle \mathcal{O}, \mathcal{M}, \mathcal{S} \rangle$ 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** $\mathcal{V} = \mathcal{M}(\mathcal{D})$.
- The set of constants that can appear in $\mathcal{V}$ consists of:
  - values obtained directly from the database, and
  - IRIs, which are constructed by applying the **iri** function to string constants and database values.

  We use $C_{\mathcal{V}}$, i.e., $C_{\mathcal{M}(\mathcal{D})}$, to denote such set of constants.

A first-order interpretation $\mathcal{I}$ of the ontology predicates and the constants in $C_{\mathcal{M}(\mathcal{D})}$ is a **model** of $\langle \mathcal{P}, \mathcal{D} \rangle$ if

- it satisfies all axioms in $\mathcal{O}$, and
- contains all facts in $\mathcal{M}(\mathcal{D})$, i.e., retrieved from $\mathcal{D}$ through $\mathcal{M}$.

Note:

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

Query answering in VKGs – 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$(\boldsymbol{q}, \mathcal{J})$ – Intuition

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

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

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

Note: Each certain answer $\vec{c}$ is a tuple of constants in $C_{\mathcal{M}(\mathcal{D})}$, but when we evaluate $\boldsymbol{q}$ over an interpretation $\mathcal{I}$, it returns tuples of elements of $\Delta^{\mathcal{I}}$. Therefore, we should actually require that $\vec{c}^{\mathcal{I}} \in \boldsymbol{q}(\mathcal{I})$, and not that $\vec{c} \in \boldsymbol{q}(\mathcal{I})$. 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 $\mathcal{P} = \langle \mathcal{O}, \mathcal{M}, \mathcal{S} \rangle$.

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 $\langle \mathcal{P}, \mathcal{D} \rangle$ = answers to $r(\vec{x})$ over $\mathcal{D}$.

For OWL 2 QL ontologies and 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.**

Computational complexity of query answering

Theorem [C., De Giacomo, et al. 2007; Poggi et al. 2008; Artale et al. 2009]

For OWL 2 QL (or *DL-Lite*) VKG instances $\langle\mathcal{P},\mathcal{D}\rangle$, with $\mathcal{P} = \langle\mathcal{O},\mathcal{M},\mathcal{S}\rangle$, **query answering** for UCQs / SPARQL queries is:

1. Very efficiently tractable, i.e., in AC^0, in the size of the database $\mathcal{D}$.
2. Efficiently tractable, , i.e., in LogSpace, in the size of the ontology $\mathcal{O}$ and the mapping $\mathcal{M}$.
3. Exponential, more precisely NP-complete, in the size of the query.

   In theory this is not bad, since this is the complexity of evaluating CQs in relational DBs.

Note: The AC^0 result is a consequence of the fact that query answering in such a setting can be reduced to evaluating a SQL query over the relational database $\mathcal{D}$.

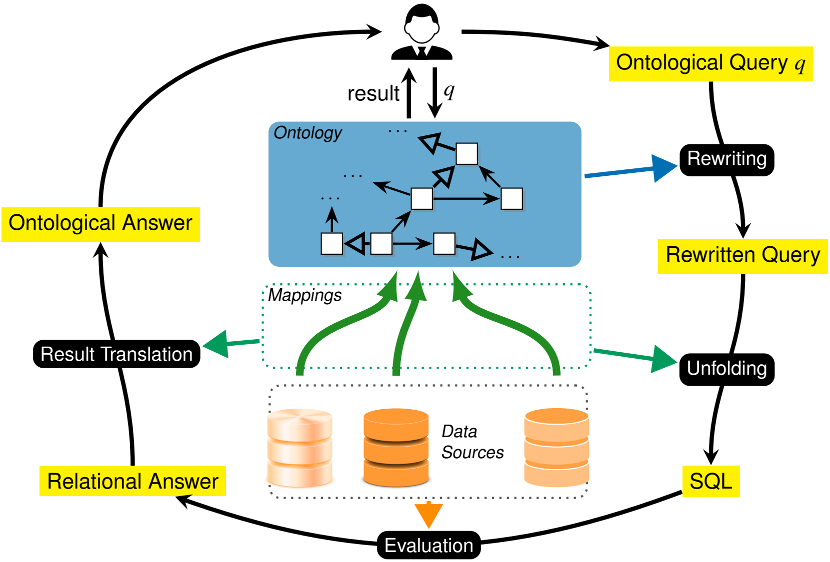
Can we go beyond *DL-Lite* and maintain the same complexity results?

Essentially no! By adding essentially any additional constructs of OWL, we lose first-order rewritability and hence these nice computational properties. [C., De Giacomo, et al. 2006, 2013]

Outline

1. Challenges in Data Access
2. Virtual Knowledge Graphs for Data Access (and Integration)
3. Optimizing Query Answering in VKGs
    - Query rewriting wrt an OWL 2 QL ontology
    - Query unfolding wrt a mapping
    - Mapping saturation
    - Optimization of query reformulation
4. Designing VKG Mappings
5. The Ontop System
6. Conclusions

Query answering via query reformulation – Conceptual framework



unibz

Query answering via query reformulation – Optimizations needed

The above conceptual framework is realized as follows.

Computing certain answers to a SPARQL query q over a VKG instance $\langle \mathcal{P}, \mathcal{D} \rangle$, with $\mathcal{P} = \langle \mathcal{O}, \mathcal{M}, \mathcal{S} \rangle$:

1. Compute the perfect rewriting of q w.r.t. $\mathcal{O}$.
2. Unfold the perfect rewriting w.r.t. the mapping $\mathcal{M}$.
3. **Optimize** the unfolded query, using database constraints.
4. Evaluate the resulting SQL query over $\mathcal{D}$.

Steps ①–③ are collectively called **query reformulation**.

We analyze now these steps more in detail.

Outline

1. Challenges in Data Access
2. Virtual Knowledge Graphs for Data Access (and Integration)
3. Optimizing Query Answering in VKGs
   - Query rewriting wrt an OWL 2 QL ontology
   - Query unfolding wrt a mapping
   - Mapping saturation
   - Optimization of query reformulation
4. Designing VKG Mappings
5. The Ontop System
6. Conclusions

Rewriting step

The rewriting Step ① deals with the knowledge encoded by the axioms of the ontology:

- hierarchies of classes and of properties;
- objects that are existentially implied by such axioms: existential reasoning.

We illustrate the need for dealing with these two aspects with two examples.

Dealing with hierarchies

Suppose that every graduate student is a student, i.e.,

$$\text{GraduateStudent} \sqsubseteq \text{Student}$$

and john is a graduate student: $\text{GraduateStudent}(\texttt{john})$.

What is the answer to the following query, asking for all students?

$$\text{q}(x) \leftarrow \text{Student}(x)$$

In SPARQL: `SELECT ?x WHERE { ?x a Student . }`

The answer should be `john`, since being a graduate student, he is also a student.

Dealing with existential reasoning

Suppose that every student is supervised by some professor, i.e.,

$$\mathsf{Student} \sqsubseteq \exists \mathsf{isSupervisedBy}.\mathsf{Professor}$$

and john is a student: $\mathsf{Student}(\mathtt{john})$.

What is the answer to the following query, asking for all individuals supervised by some professor?

$$\mathsf{q}(x) \leftarrow \mathsf{isSupervisedBy}(x, y),\ \mathsf{Professor}(y)$$

In SPARQL: `SELECT ?x WHERE { ?x isSupervisedBy [ a Professor ] . }`

The answer should be `john`, even though we don't know who is John's supervisor (under existential reasoning).

The query rewriting algorithm

The **query rewriting** algorithm takes into account hierarchies and existential reasoning, by "compiling" the axioms of the ontology into the query.

Example

Consider the ontology axioms:

$$\begin{aligned} \text{Student} &\sqsubseteq \exists \text{isSupervisedBy}.\text{Professor} \\ \text{GraduateStudent} &\sqsubseteq \text{Student} \end{aligned}$$

Using these axioms, the rewriting algorithm rewrites the query

$$\text{q}(x) \leftarrow \text{isSupervisedBy}(x, y),\ \text{Professor}(y)$$

into a union of conjunctive queries (or a SPARQL union query):

$$\begin{aligned} \text{q}(x) &\leftarrow \text{isSupervisedBy}(x, y),\ \text{Professor}(y) \\ \text{q}(x) &\leftarrow \text{Student}(x) \\ \text{q}(x) &\leftarrow \text{GraduateStudent}(x) \end{aligned}$$

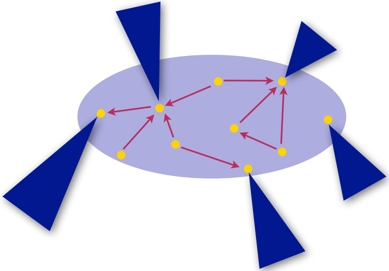
Therefore, over the data $\text{Student}(\texttt{john})$, the rewritten query returns `john` as an answer.

Note: In *Ontop*, existential reasoning needs to be switched on explicitly, since it affects performance.

Query rewriting and canonical model

Canonical model

Every consistent *DL-Lite* / OWL 2 QL KB $\mathcal{K} = \langle \mathcal{O}, \mathcal{A} \rangle$ has a **canonical model** $\mathcal{I}_{\mathcal{K}}$, which **gives the right answers to all CQs**, i.e., $\text{cert}(q, \mathcal{K}) = \text{ans}(q, \mathcal{I}_{\mathcal{K}})$



- The core part can be handled by **saturating the mapping**.
- The anonymous part can be handled by **tree-witness rewriting**.

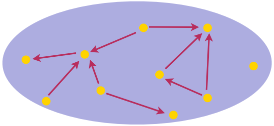
unibz

Query rewriting and canonical model

Canonical model

Every consistent *DL-Lite* / OWL 2 QL KB $\mathcal{K} = \langle \mathcal{O}, \mathcal{A} \rangle$ has a **canonical model** $\mathcal{I}_{\mathcal{K}}$, which **gives the right answers to all CQs**, i.e., $\text{cert}(q, \mathcal{K}) = \text{ans}(q, \mathcal{I}_{\mathcal{K}})$

Core
individuals from $\mathcal{A}$

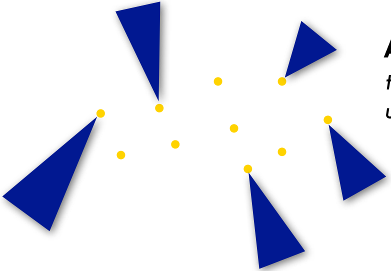


- The core part can be handled by **saturating the mapping**.
- The anonymous part can be handled by **tree-witness rewriting**.

Query rewriting and canonical model

Canonical model

Every consistent *DL-Lite* / OWL 2 QL KB $\mathcal{K} = \langle \mathcal{O}, \mathcal{A} \rangle$ has a **canonical model** $\mathcal{I}_{\mathcal{K}}$, which **gives the right answers to all CQs**, i.e., $\mathrm{cert}(q, \mathcal{K}) = \mathrm{ans}(q, \mathcal{I}_{\mathcal{K}})$



Anonymous part

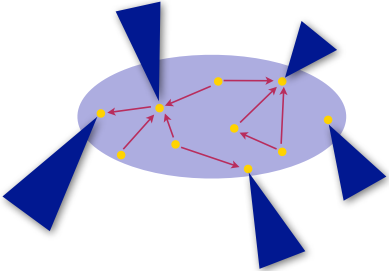
trees rooted at individuals, using unnamed objects

- The core part can be handled by **saturating the mapping**.
- The anonymous part can be handled by **tree-witness rewriting**.

Query rewriting and canonical model

Canonical model

Every consistent *DL-Lite* / OWL 2 QL KB $\mathcal{K} = \langle \mathcal{O}, \mathcal{A} \rangle$ has a **canonical model** $\mathcal{I}_{\mathcal{K}}$, which **gives the right answers to all CQs**, i.e., $\mathrm{cert}(q, \mathcal{K}) = \mathrm{ans}(q, \mathcal{I}_{\mathcal{K}})$

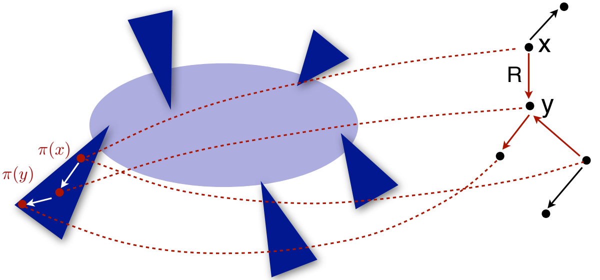


- The core part can be handled by **saturating the mapping**.
- The anonymous part can be handled by **tree-witness rewriting**.

Query rewriting and canonical model

Canonical model

Every consistent *DL-Lite* / OWL 2 QL KB $\mathcal{K} = \langle \mathcal{O}, \mathcal{A} \rangle$ has a **canonical model** $\mathcal{I}_{\mathcal{K}}$, which **gives the right answers to all CQs**, i.e., cert$(q, \mathcal{K})$ = ans$(q, \mathcal{I}_{\mathcal{K}})$



- The core part can be handled by **saturating the mapping**.
- The anonymous part can be handled by **tree-witness rewriting**.

unibz

The *PerfectRef* algorithm for query rewriting

We do not describe here the tree-witness rewriting algorithm, which is rather involved.

Instead, we describe *PerfectRef*, a simple query rewriting algorithm that maintains a set of queries and applies over them two types of transformations:

- rewriting steps that involve inclusion assertions of the ontology, and
- unification of query atoms.

These transformations are applied repeatedly until saturation, i.e., until the set of queries does not change anymore.

Given as input a (core) SPARQL query q, *PerfectRef* computes its **perfect rewriting**, which is still a SPARQL query (involving `UNION`).

Note: Disjointness assertions play a role in ontology satisfiability, but can be ignored during query rewriting. (This is called **separability**.)

Query rewriting step: Basic idea

Intuition: an inclusion assertion corresponds to a logic programming rule.

Basic rewriting step:

When an atom in the query unifies with the head of the rule, generate a new query by substituting the atom with the body of the rule.

We say that the inclusion assertion **applies to** the atom.

Example

The inclusion assertion $\text{Professor} \sqsubseteq \text{Teacher}$
corresponds to the logic programming rule $\text{Teacher}(z) \leftarrow \text{Professor}(z)$.

Consider the query $\text{q}(x) \leftarrow \text{Teacher}(x)$.

By applying the inclusion assertion to the atom $\text{Teacher}(x)$, we generate:

$$\text{q}(x) \leftarrow \text{Professor}(x).$$

This query is added to the input query, and contributes to the perfect rewriting.

Query rewriting (cont'd)

Example

Consider the query $\mathsf{q}(x) \leftarrow \mathsf{teaches}(x,y), \mathsf{Course}(y)$

and the inclusion assertion $\exists\mathsf{teaches}^{-} \sqsubseteq \mathsf{Course}$
as a logic programming rule: $\mathsf{Course}(z_2) \leftarrow \mathsf{teaches}(z_1,z_2)$.

The inclusion applies to $\mathsf{Course}(y)$, and we add to the rewriting the query

$$\mathsf{q}(x) \leftarrow \mathsf{teaches}(x,y), \mathsf{teaches}(z_1,y).$$

Example

Consider now the query $\mathsf{q}(x) \leftarrow \mathsf{teaches}(x,y)$

and the inclusion assertion $\mathsf{Professor} \sqsubseteq \exists\mathsf{teaches}$
as a logic programming rule: $\mathsf{teaches}(z,f(z)) \leftarrow \mathsf{Professor}(z)$.

The inclusion applies to $\mathsf{teaches}(x,y)$, and we add to the rewriting the query

$$\mathsf{q}(x) \leftarrow \mathsf{Professor}(x).$$

Query rewriting – Constants

Example

Conversely, for the query $\mathsf{q}(x) \leftarrow \mathsf{teaches}(x, \text{"}\mathtt{databases}\text{"})$

and the same inclusion assertion as before $\mathsf{Professor} \sqsubseteq \exists\mathsf{teaches}$
as a logic programming rule: $\mathsf{teaches}(z, f(z)) \leftarrow \mathsf{Professor}(z)$

$\mathsf{teaches}(x, \text{"}\mathtt{databases}\text{"})$ does not unify with $\mathsf{teaches}(z, f(z))$, since the **skolem term** $f(z)$ in the head of the rule **does not unify** with the constant "$\mathtt{databases}$".
Remember: We adopt the **unique name assumption**.

We say that the inclusion does **not** apply to the atom $\mathsf{teaches}(x, \text{"}\mathtt{databases}\text{"})$.

Example

The same holds for the following query, where y is **distinguished**, since unifying $f(z)$ with y would correspond to returning a skolem term as answer to the query:

$$\mathsf{q}(x, y) \leftarrow \mathsf{teaches}(x, y).$$

Query rewriting – Join variables

An analogous behavior to the one with constants and with distinguished variables holds when the atom contains **join variables** that would have to be unified with skolem terms.

Example

Consider the query $\mathsf{q}(x) \leftarrow \mathsf{teaches}(x, y), \mathsf{Course}(y)$

and the inclusion assertion $\mathsf{Professor} \sqsubseteq \exists\mathsf{teaches}$
as a logic programming rule: $\mathsf{teaches}(z, f(z)) \leftarrow \mathsf{Professor}(z)$.

The inclusion assertion above does **not** apply to the atom $\mathsf{teaches}(x, y)$.

Query rewriting – Reduce step

Example

Consider now the query $\mathsf{q}(x) \leftarrow \mathsf{teaches}(x,y), \mathsf{teaches}(z,y)$

and the inclusion assertion $\mathsf{Professor} \sqsubseteq \exists\mathsf{teaches}$
as a logic rule: $\mathsf{teaches}(z, f(z)) \leftarrow \mathsf{Professor}(z)$.

This inclusion assertion does not apply to $\mathsf{teaches}(x,y)$ or $\mathsf{teaches}(z,y)$, since y is in join, and we would again introduce the skolem term in the rewritten query.

Example

However, we can transform the above query by unifying the atoms $\mathsf{teaches}(x,y)$ and $\mathsf{teaches}(z,y)$. This rewriting step is called **reduce**, and produces the query

$$\mathsf{q}(x) \leftarrow \mathsf{teaches}(x,y).$$

Now, we can apply the inclusion above, and add to the rewriting the query

$$\mathsf{q}(x) \leftarrow \mathsf{Professor}(x).$$

Query rewriting – Summary

To compute the perfect rewriting of a query q with respect to an ontology $\mathcal{O}$, start from q, iteratively get a CQ q' to be processed, and do one of the following:

- Apply to some atom of q' an inclusion assertion in $\mathcal{O}$ as follows:

$$\begin{array}{rccc}
C_1 \sqsubseteq C_2 & \ldots, C_2(x), \ldots & \rightsquigarrow & \ldots, C_1(x), \ldots \\
\exists P \sqsubseteq C & \ldots, C(x), \ldots & \rightsquigarrow & \ldots, P(x, \_), \ldots \\
\exists P^- \sqsubseteq C & \ldots, C(x), \ldots & \rightsquigarrow & \ldots, P(\_, x), \ldots \\
C \sqsubseteq \exists P & \ldots, P(x, \_), \ldots & \rightsquigarrow & \ldots, C(x), \ldots \\
C \sqsubseteq \exists P^- & \ldots, P(\_, x), \ldots & \rightsquigarrow & \ldots, C(x), \ldots \\
\exists P_1 \sqsubseteq \exists P_2 & \ldots, P_2(x, \_), \ldots & \rightsquigarrow & \ldots, P_1(x, \_), \ldots \\
P_1 \sqsubseteq P_2 & \ldots, P_2(x, y), \ldots & \rightsquigarrow & \ldots, P_1(x, y), \ldots \\
P_1 \sqsubseteq P_2^- & \ldots, P_2(x, y), \ldots & \rightsquigarrow & \ldots, P_1(y, x), \ldots
\end{array}$$

  ('_' denotes a variable that appears only once)

- Choose two atoms of q' that unify, and apply the unifier to q'.

After each rewriting/unification step, the obtained query is added to the queries still to be processed.

Note: Unifying atoms can make rules applicable that were not so before, and is required for completeness of the method [C., De Giacomo, et al. 2007].

The UCQ resulting from this process is the **perfect rewriting** q_r of q w.r.t. the ontology $\mathcal{O}$.

Query rewriting algorithm

```
Algorithm PerfectRef($Q, \mathcal{O}_P$)
Input: union of conjunctive queries $Q$, set $\mathcal{O}_P$ of DL-Lite / OWL 2 QL positive inclusion assertions
Output: union of conjunctive queries $PR$
$PR := Q$;
repeat
  $PR' := PR$;
  for each $q \in PR'$ do
    for each $g$ in $q$ do
      for each inclusion assertion $I$ in $\mathcal{O}_P$ do
        if $I$ is applicable to $g$ then $PR := PR \cup \{ ApplyPI(q, g, I) \}$;
    for each $g_1, g_2$ in $q$ do
      if $g_1$ and $g_2$ unify then $PR := PR \cup \{\tau(Reduce(q, g_1, g_2))\}$;
until $PR' = PR$;
return $PR$
```

Observations:

- Termination follows from having only finitely many different rewritings.
- Disjointness assertions and functionalities do not play any role in the rewriting of the query.

Query answering in *DL-Lite* – Example

Ontology:

$$\begin{aligned} \text{Professor} &\sqsubseteq \text{Teacher} \\ \text{Teacher} &\sqsubseteq \exists\text{teaches} \\ \exists\text{teaches}^- &\sqsubseteq \text{Course} \end{aligned}$$

Corresponding rules:

$$\begin{aligned} \text{Teacher}(z) &\leftarrow \text{Professor}(z) \\ \text{teaches}(z, f(z)) &\leftarrow \text{Teacher}(z) \\ \text{Course}(z) &\leftarrow \text{teaches}(w, z) \end{aligned}$$

Query: $\text{q}(x) \leftarrow \text{teaches}(x, y), \text{Course}(y)$

Perfect rewriting: $\text{q}(x) \leftarrow \text{teaches}(x, y), \text{Course}(y)$

$$\begin{aligned} \text{q}(x) &\leftarrow \text{teaches}(x, y), \text{teaches}(\_, y) \\ \text{q}(x) &\leftarrow \text{teaches}(x, \_) \\ \text{q}(x) &\leftarrow \text{Teacher}(x) \\ \text{q}(x) &\leftarrow \text{Professor}(x) \end{aligned}$$

ABox: $\text{teaches}(\texttt{jim}, \texttt{databases})$ $\text{Professor}(\texttt{jim})$

$\text{teaches}(\texttt{julia}, \texttt{security})$ $\text{Teacher}(\texttt{nicole})$

Evaluating the perfect rewriting over the ABox (seen as a DB) produces as answer $\{\texttt{jim}, \texttt{julia}, \texttt{nicole}\}$.

Query answering in *DL-Lite* – An interesting example

TBox: $\text{Person} \sqsubseteq \exists\text{hasFather}$
$\exists\text{hasFather}^- \sqsubseteq \text{Person}$

ABox: $\text{Person}(\texttt{john})$

Query: $\text{q}(x) \leftarrow \text{Person}(x), \text{hasFather}(x, y_1), \text{hasFather}(y_1, y_2), \text{hasFather}(y_2, y_3)$

$\text{q}(x) \leftarrow \text{Person}(x), \text{hasFather}(x, y_1), \text{hasFather}(y_1, y_2), \text{hasFather}(y_2, \_)$

$\Downarrow$ **Apply** $\text{Person} \sqsubseteq \exists\text{hasFather}$ to the atom $\text{hasFather}(y_2, \_)$

$\text{q}(x) \leftarrow \text{Person}(x), \text{hasFather}(x, y_1), \text{hasFather}(y_1, y_2), \text{Person}(y_2)$

$\Downarrow$ **Apply** $\exists\text{hasFather}^- \sqsubseteq \text{Person}$ to the atom $\text{Person}(y_2)$

$\text{q}(x) \leftarrow \text{Person}(x), \text{hasFather}(x, y_1), \text{hasFather}(y_1, y_2), \text{hasFather}(\_, y_2)$

$\Downarrow$ **Unify** atoms $\text{hasFather}(y_1, y_2)$ and $\text{hasFather}(\_, y_2)$

$\text{q}(x) \leftarrow \text{Person}(x), \text{hasFather}(x, y_1), \text{hasFather}(y_1, y_2)$

$\Downarrow$

$\cdots$

$\text{q}(x) \leftarrow \text{Person}(x), \text{hasFather}(x, \_)$

$\Downarrow$ **Apply** $\text{Person} \sqsubseteq \exists\text{hasFather}$ to the atom $\text{hasFather}(x, \_)$

$\text{q}(x) \leftarrow \text{Person}(x)$

Exponential blowup in the rewriting

Even with a flat hierarchy of classes in which a single inclusion assertion can be applied to each atom of a query $\boldsymbol{q}$, the rewriting may contain an number of CQs that is exponential in the length of $\boldsymbol{q}$.

Consider a query: $\boldsymbol{q}(x) \leftarrow C_1^1(x),\ C_2^1(x),\ \ldots,\ C_n^1(x)$

and the ontology: $\mathcal{O} = \{\ C_1^2 \sqsubseteq C_1^1,\ C_2^2 \sqsubseteq C_2^1,\ \ldots,\ C_n^2 \sqsubseteq C_n^1\ \}$

Each atom $C_i^1(x)$ in $\boldsymbol{q}$ can either stay as is, or we can apply to it the inclusion assertion $C_i^2 \sqsubseteq C_i^1$, and generate a new CQ in which $C_i^1(x)$ is replaced by $C_i^2(x)$.

Hence, in the rewriting we have one CQ

$$\boldsymbol{q}(x) \leftarrow C_1^{j_1}(x),\ C_2^{j_2}(x),\ \ldots,\ C_n^{j_n}(x)$$

for each possible combination of $j_i \in \{1, 2\}$, for $i \in \{1, \ldots, n\}$.

Hence, the rewriting of $\boldsymbol{q}$ with respect to $\mathcal{O}$ contains 2^n **CQs**.

Outline

1. Challenges in Data Access
2. Virtual Knowledge Graphs for Data Access (and Integration)
3. Optimizing Query Answering in VKGs
   - Query rewriting wrt an OWL 2 QL ontology
   - Query unfolding wrt a mapping
   - Mapping saturation
   - Optimization of query reformulation
4. Designing VKG Mappings
5. The Ontop System
6. Conclusions

Query unfolding

We consider now Step ② of reformulation, i.e., the unfolding w.r.t. the mapping $\mathcal{M}$.

In principle, we have two approaches to exploit the mapping:

- bottom-up approach: simpler, but typically less efficient
- top-down approach: more sophisticated, but also more efficient

Both approaches require to first **split** the set of atoms in the target queries of the mapping assertions into the constituent atoms.

Note: In the following, to make notation more compact, we represent an IRI-template of the form

$$\texttt{:xxx/\{}v_1\texttt{\}/\{}v_2\texttt{\}/}\cdots\texttt{/\{}v_n\texttt{\}}$$

more compactly as

$$\mathbf{xxx}(v_1, \ldots, v_n).$$

Splitting of mappings

A mapping assertion $\Phi \rightsquigarrow \Psi$, where the target query Ψ is constituted by the atoms $X_1, \ldots, X_k$, can be split into k mapping assertions:

$$\Phi \rightsquigarrow X_1 \qquad \cdots \qquad \Phi \rightsquigarrow X_k$$

This is possible, since Ψ does not contain non-distinguished variables.

Example

m_1: `SELECT pcode, acode, aname FROM ACTOR` $\rightsquigarrow$ Play(**pl**($pcode$)),
Actor(**act**($acode$)),
name(**act**($acode$), $aname$),
actsIn(**act**($acode$), **pl**($pcode$))

is split into

m_1^1: `SELECT pcode, acode, aname FROM ACTOR` $\rightsquigarrow$ Play(**pl**($pcode$))
m_1^2: `SELECT pcode, acode, aname FROM ACTOR` $\rightsquigarrow$ Actor(**act**($acode$))
m_1^3: `SELECT pcode, acode, aname FROM ACTOR` $\rightsquigarrow$ name(**act**($acode$), $aname$)
m_1^4: `SELECT pcode, acode, aname FROM ACTOR` $\rightsquigarrow$ actsIn(**act**($acode$), **pl**($pcode$))

Bottom-up approach to deal with mappings: Materialization

Consists in a straightforward application of the mappings to the data:

1. Propagate the data from $\mathcal{D}$ through $\mathcal{M}$, **materializing** the RDF graph $\mathcal{V} = \mathcal{M}(\mathcal{D})$ (the constants in such an RDF graph are values and object terms obtained from the database values).
2. Apply to $\mathcal{V}$ and to the ontology $\mathcal{O}$, the query answering algorithm (based on query rewriting) developed for *DL-Lite* / OWL 2 QL.

This approach has several drawbacks:

- The technique is no more AC^0 in the size of the data, since the RDF graph $\mathcal{V}$ to materialize is in general polynomial in the size of the data.
- $\mathcal{V}$ may be very large, and thus it may be infeasible to actually materialize it.
- Freshness of $\mathcal{V}$ with respect to the underlying data source(s) may be an issue, and one would need to propagate source updates (cf. Data Warehousing).

Top-down approach to deal with mappings: Unfolding

The top-down approach is realized by computing from the (rewritten) query q_{r} a new query q_{unf}, by **unfolding** q_{r} using (the split version of) the mappings $\mathcal{M}$.

Consider the mapping assertions $\Phi_i \rightsquigarrow \Psi_i$.

- Essentially, each atom in q_{r} that unifies with an atom in some Ψ_i is substituted with the corresponding query Φ_i over the database.
- The unfolded query q_{unf} is such that for each database $\mathcal{D}$ we have that:

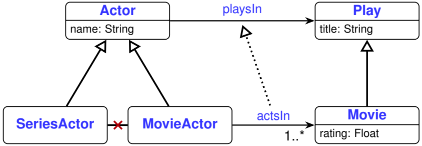
$$q_{\mathsf{unf}}(\mathcal{D}) \;=\; \mathit{Eval}_{\mathsf{CWA}}(q_{\mathsf{r}}, \mathcal{M}(\mathcal{D})).$$

Unfolding

To unfold a query q_r with respect to a set $\mathcal{M}$ of mapping assertions:

1. For each non-split mapping assertion $\Phi_i(\vec{x}) \rightsquigarrow \Psi_i(\vec{t}, \vec{y})$:
    1. Introduce a **view symbol** Aux_i of arity equal to that of Φ_i.
    2. Add a **view definition** $\mathsf{Aux}_i(\vec{x}) \leftarrow \Phi_i(\vec{x})$.
2. For each split version $\Phi_i(\vec{x}) \rightsquigarrow X_i^j(\vec{t}, \vec{y})$ of a mapping assertion, introduce a **clause** $X_i^j(\vec{t}, \vec{y}) \leftarrow \mathsf{Aux}_i(\vec{x})$.
3. Obtain from q_r in all possible ways queries q_{aux} defined over the view symbols Aux_i as follows:
    1. Find a most general unifier ϑ that unifies each atom $X(\vec{z})$ in the body of q_r with the head of a clause $X(\vec{t}, \vec{y}) \leftarrow \mathsf{Aux}_i(\vec{x})$.
    2. Substitute each atom $X(\vec{z})$ with $\vartheta(\mathsf{Aux}_i(\vec{x}))$, i.e., with the body the unified clause to which the unifier ϑ is applied.
4. The unfolded query q_{unf} is the **union** of all queries q_{aux}, together with the view definitions for the predicates Aux_i appearing in q_{aux}.

Unfolding – Example



m_1:
```
SELECT pcode, acode, aname
FROM ACTOR
```
$\rightsquigarrow$ Play(**pl**($pcode$)),
Actor(**act**($acode$)),
name(**act**($acode$), $aname$),
actsIn(**act**($acode$), **pl**($pcode$))

m_2:
```
SELECT mcode, acode, mtitle
FROM MOVIE M, ACTOR A
WHERE M.mcode = A.pcode
  AND M.type = "m"
```
$\rightsquigarrow$ Movie(**pl**($mcode$)),
playsIn(**act**($acode$),
pl($mcode$)),
title(**pl**($mcode$), $mtitle$)

We define a view Aux$_i$ for the source query of each mapping m_i.

For each (split) mapping assertion, we introduce a clause:

$$\begin{aligned}
\mathsf{Play}(\textbf{pl}(pcode)) &\leftarrow \mathsf{Aux}_1(pcode, \_, \_)\\
\mathsf{Actor}(\textbf{act}(acode)) &\leftarrow \mathsf{Aux}_1(\_, acode, \_)\\
\mathsf{name}(\textbf{act}(acode), aname) &\leftarrow \mathsf{Aux}_1(\_, acode, aname)\\
\mathsf{actsIn}(\textbf{act}(acode), \textbf{pl}(pcode)) &\leftarrow \mathsf{Aux}_1(pcode, acode, \_)\\
\mathsf{Movie}(\textbf{pl}(mcode)) &\leftarrow \mathsf{Aux}_2(mcode, \_, \_)\\
\mathsf{playsIn}(\textbf{act}(acode), \textbf{pl}(mcode)) &\leftarrow \mathsf{Aux}_2(mcode, acode, \_)\\
\mathsf{title}(\textbf{pl}(mcode), mtitle) &\leftarrow \mathsf{Aux}_2(mcode, \_, mtitle)
\end{aligned}$$

Unfolding – Example (cont'd)

Query over the ontology: Actors with their name who act in a movie whose title is `"The Matrix"`:

$$q(a,n) \leftarrow \mathsf{Actor}(a), \mathsf{name}(a,n), \mathsf{actsIn}(a,p), \mathsf{Movie}(p), \mathsf{title}(p, \texttt{"The Matrix"})$$

$$\begin{aligned}
\mathsf{Actor}(\mathbf{act}(acode)) &\leftarrow \mathsf{Aux}_1(\_, acode, \_)\\
\mathsf{name}(\mathbf{act}(acode), aname) &\leftarrow \mathsf{Aux}_1(\_, acode, aname)\\
\mathsf{actsIn}(\mathbf{act}(acode), \mathbf{pl}(pcode)) &\leftarrow \mathsf{Aux}_1(pcode, acode, \_)\\
\mathsf{Movie}(\mathbf{pl}(mcode)) &\leftarrow \mathsf{Aux}_2(mcode, \_, \_)\\
\mathsf{title}(\mathbf{pl}(mcode), mtitle) &\leftarrow \mathsf{Aux}_2(mcode, \_, mtitle)
\end{aligned}$$

A unifier ϑ between the atoms in q and the clause heads is:

$$\vartheta(a) = \mathbf{act}(acode) \qquad \vartheta(n) = aname$$
$$\vartheta(p) = \mathbf{pl}(pcode) \qquad \vartheta(mcode) = pcode \qquad \vartheta(mtitle) = \texttt{"The Matrix"}$$

After applying ϑ to q, we obtain:

$$\begin{aligned}
q(\mathbf{act}(acode), aname) \leftarrow\ & \mathsf{Actor}(\mathbf{act}(acode)),\ \mathsf{name}(\mathbf{act}(acode), aname),\ \mathsf{actsIn}(\mathbf{act}(acode), \mathbf{pl}(pcode)),\\
& \mathsf{Movie}(\mathbf{pl}(pcode)),\ \mathsf{title}(\mathbf{pl}(pcode), \texttt{"The Matrix"})
\end{aligned}$$

Substituting the atoms with the bodies of the clauses (after having applied the unifier), we obtain:

$$\begin{aligned}
q(\mathbf{act}(acode), aname) \leftarrow\ & \mathsf{Aux}_1(\_, acode, \_),\ \mathsf{Aux}_1(\_, acode, aname),\ \mathsf{Aux}_1(pcode, acode, \_),\\
& \mathsf{Aux}_2(pcode, \_, \_),\ \mathsf{Aux}_2(pcode, \_, \texttt{"The Matrix"})
\end{aligned}$$

Exponential blowup in the unfolding

When there are multiple mapping assertions for each atom, the unfolded query may be exponential in the original one.

Consider a query: $\boldsymbol{q}(y) \leftarrow C_1(y), C_2(y), \ldots, C_n(y)$

and the mappings: $m_i^1 \colon \Phi_i^1(x) \rightsquigarrow C_i(\textbf{iri}(x))$ (for $i \in \{1, \ldots, n\}$)
$m_i^2 \colon \Phi_i^2(x) \rightsquigarrow C_i(\textbf{iri}(x))$

We add the view definitions: $\mathsf{Aux}_i^j(x) \leftarrow \Phi_i^j(x)$
and introduce the clauses: $C_i(\textbf{iri}(x)) \leftarrow \mathsf{Aux}_i^j(x)$ (for $i \in \{1, \ldots, n\}, j \in \{1, 2\}$).

There is a single unifier, namely $\vartheta(y) = \textbf{iri}(x)$, but each atom $C_i(y)$ in the query unifies with the head of two clauses.

Hence, we obtain one unfolded query

$$\boldsymbol{q}(\textbf{iri}(x)) \leftarrow \mathsf{Aux}_1^{j_1}(x), \mathsf{Aux}_2^{j_2}(x), \ldots, \mathsf{Aux}_n^{j_n}(x)$$

for each possible combination of $j_i \in \{1, 2\}$, for $i \in \{1, \ldots, n\}$.
Hence, we obtain 2^n **unfolded queries**.

Implementation of top-down approach to query answering

To implement the top-down approach, we need to generate an SQL query.

We can follow different strategies:

1. Substitute each view predicate in the unfolded queries with the corresponding SQL query over the source:
   - \+ joins are performed on the DB attributes, hence can be done efficiently, e.g., by exploiting indexes;
   - \+ does not generate doubly nested queries;
   - – the number of unfolded queries may be exponential.

2. Construct for each atom in the original query a new view. This view takes the union of all SQL queries corresponding to the view predicates, and constructs also the IRIs based on the IRI templates:
   - \+ avoids exponential blow-up of the resulting query, since the union (of the queries coming from multiple mappings) is done before the joins;
   - – joins are performed on IRIs, i.e., on terms built using string concatenation, hence are highly inefficient;
   - – generates doubly nested queries, which per se the database has difficulty in optimizing.

Which method is better, depends on various parameters, and there is no definitive answer.
In general, one needs a mixed approach that applies different strategies to different parts of the query.

Outline

1. Challenges in Data Access
2. Virtual Knowledge Graphs for Data Access (and Integration)
3. Optimizing Query Answering in VKGs
    - Query rewriting wrt an OWL 2 QL ontology
    - Query unfolding wrt a mapping
    - Mapping saturation
    - Optimization of query reformulation
4. Designing VKG Mappings
5. The Ontop System
6. Conclusions

Contributions of rewriting and unfolding

- We are interested in computing certain answers to SPARQL queries over a VKG instance $\langle \mathcal{P}, \mathcal{D} \rangle$, with $\mathcal{P} = \langle \mathcal{O}, \mathcal{M}, \mathcal{S} \rangle$.
- In practice, by computing the rewriting q_{r} of q w.r.t. $\mathcal{O}$ and its unfolding w.r.t. $\mathcal{M}$, the resulting query q_{unf} might become very large, and costly to execute over $\mathcal{D}$.

Let us consider the contributions of rewriting and unfolding to the query answers:

- In principle, evaluating the unfolding q_{unf} (of q_{r} w.r.t. $\mathcal{M}$) over $\mathcal{D}$, gives the same result as evaluating q_{r} over the RDF graph $\mathcal{V} = \mathcal{M}(\mathcal{D})$ extracted through the mapping $\mathcal{M}$ from the data $\mathcal{D}$.
- Instead, the impact of the rewriting on the query answers consists of two components:
  1. the rewriting w.r.t. class and property hierarchies, i.e., $C_1 \sqsubseteq C_2$, $P_1 \sqsubseteq P_2$;
  2. the rewriting taking into account existential reasoning, i.e., $C \sqsubseteq \exists R$, $C_1 \sqsubseteq \exists R.C_2$.

Note: Component ① corresponds to computing the saturation $\mathcal{V}_{\mathsf{sat}}$ of $\mathcal{V}$ w.r.t. class and property hierarchies, while component ② can be handled only through rewriting.

Tree-witness rewriting and saturated mapping

We want to avoid materializing $\mathcal{V}$ and $\mathcal{V}_{\text{sat}}$, but also want to avoid computing the query rewriting w.r.t. class and property hierarchies.

Therefore we proceed as follows:

1. We rewrite q only w.r.t. the inclusion assertions that cause existential reasoning (i.e., $C \sqsubseteq \exists R$, $C_1 \sqsubseteq \exists R.C_2$).
   $\leadsto$ **tree-witness rewriting** q_{tw} [Kikot et al. 2012]
2. We use instead class and property hierarchies (i.e., $C_1 \sqsubseteq C_2$, $P_1 \sqsubseteq P_2$) to enrich the mapping $\mathcal{M}$.
   $\leadsto$ **saturated mapping** $\mathcal{M}_{\text{sat}}$ [Rodriguez-Muro et al. 2013; Kontchakov, Rezk, et al. 2014]
3. We unfold the tree-witness rewriting q_{tw} w.r.t. the saturated mapping $\mathcal{M}_{\text{sat}}$.

It is possible to show that the resulting query is equivalent to the perfect rewriting q_{r} (as obtained, e.g., through ordinary rewriting w.r.t. $\mathcal{O}$ and unfolding w.r.t. $\mathcal{M}$).

For more details, we refer also to [Kontchakov & Zakharyaschev 2014].

Saturated mapping

Intuitively, the **saturated mapping** $\mathcal{M}_{\text{sat}}$ is obtained as the composition of $\mathcal{M}$ and the ontology $\mathcal{O}$.

For each mapping assertion in $\mathcal{M}$	and each TBox assertion in $\mathcal{O}$	we add a mapping assertion to $\mathcal{M}_{\text{sat}}$
$\Phi(x) \leadsto C_1(\textbf{iri}(x))$	$C_1 \sqsubseteq C_2$	$\Phi(x) \leadsto C_2(\textbf{iri}(x))$
$\Phi(x,y) \leadsto P(\textbf{iri}_1(x), \textbf{iri}_2(y))$	$\exists P \sqsubseteq C_1$	$\Phi(x,y) \leadsto C_1(\textbf{iri}_1(x))$
$\Phi(x,y) \leadsto P(\textbf{iri}_1(x), \textbf{iri}_2(y))$	$\exists P^- \sqsubseteq C_2$	$\Phi(x,y) \leadsto C_2(\textbf{iri}_2(y))$
$\Phi(x,y) \leadsto P_1(\textbf{iri}_1(x), \textbf{iri}_2(y))$	$P_1 \sqsubseteq P_2$	$\Phi(x,y) \leadsto P_2(\textbf{iri}_1(x), \textbf{iri}_2(y))$

Due to saturation, $\mathcal{M}_{\text{sat}}$ will contain at most $|\mathcal{O}| \cdot |\mathcal{M}|$ many mappings.

Note: The saturated mapping has also been called **T-mapping** in the literature.

Saturated mapping – Exercise

Ontology $\mathcal{O}$

$\textsf{Student} \sqsubseteq \textsf{Person}$
$\textsf{PostDoc} \sqsubseteq \textsf{Faculty}$
$\textsf{Professor} \sqsubseteq \textsf{Faculty}$
$\exists\textsf{teaches} \sqsubseteq \textsf{Faculty}$
$\textsf{Faculty} \sqsubseteq \textsf{Person}$

User-defined mapping assertions $\mathcal{M}$

$\texttt{student}(scode, fn, ln) \rightsquigarrow \textsf{Student}(\textbf{iri1}(scode))$ (1)
$\texttt{academic}(acode, fn, ln, pos),\ pos = 9 \rightsquigarrow \textsf{PostDoc}(\textbf{iri2}(acode))$ (2)
$\texttt{academic}(acode, fn, ln, pos),\ pos = 2 \rightsquigarrow \textsf{Professor}(\textbf{iri2}(acode))$ (3)
$\texttt{teaching}(course, acode) \rightsquigarrow \textsf{teaches}(\textbf{iri2}(acode), \textbf{iri3}(course))$ (4)
$\texttt{academic}(acode, fn, ln, pos) \rightsquigarrow \textsf{Faculty}(\textbf{iri2}(acode))$ (5)

By **saturating the mapping**, we obtain $\mathcal{M}_{\textsf{sat}}$, containing additional mapping assertions for the classes Faculty and Person.

$\texttt{student}(scode, fn, ln) \rightsquigarrow \textsf{Person}(\textbf{iri1}(scode))$ (6)
$\texttt{academic}(acode, fn, ln, pos),\ pos = 9 \rightsquigarrow \textsf{Faculty}(\textbf{iri2}(acode))$ (7)
$\texttt{academic}(acode, fn, ln, pos),\ pos = 9 \rightsquigarrow \textsf{Person}(\textbf{iri2}(acode))$ (8)
$\texttt{academic}(acode, fn, ln, pos),\ pos = 2 \rightsquigarrow \textsf{Faculty}(\textbf{iri2}(acode))$ (9)
$\texttt{academic}(acode, fn, ln, pos),\ pos = 2 \rightsquigarrow \textsf{Person}(\textbf{iri2}(acode))$ (10)
$\texttt{academic}(acode, fn, ln, pos) \rightsquigarrow \textsf{Person}(\textbf{iri2}(acode))$ (11)
$\texttt{teaching}(course, acode) \rightsquigarrow \textsf{Faculty}(\textbf{iri2}(acode))$ (12)
$\texttt{teaching}(course, acode) \rightsquigarrow \textsf{Person}(\textbf{iri2}(acode))$ (13)

Properties of saturated mappings

H-complete RDF graph

An RDF graph $\mathcal{G}$ is **H-complete** w.r.t. an ontology $\mathcal{O}$, if, for every RDF triple (s, p, o), we have:

$$\langle \mathcal{O}, \mathcal{G} \rangle \models (s, p, o) \qquad \text{iff} \qquad (s, p, o) \in \mathcal{G}$$

The **saturation** $\mathcal{G}_{\mathsf{sat}}$ of $\mathcal{G}$ w.r.t. $\mathcal{O}$ is the smallest RDF graph that contains $\mathcal{G}$ and is H-complete w.r.t. $\mathcal{O}$.

Intuitively, $\mathcal{G}_{\mathsf{sat}}$ is obtained from $\mathcal{G}$ by applying the class and property inclusions of $\mathcal{O}$, but without introducing new nodes.

Relationship between the saturated mapping $\mathcal{M}_{\mathsf{sat}}$ and the saturation of $\mathcal{M}(\mathcal{D})$

- We have that $\mathcal{M}_{\mathsf{sat}}(\mathcal{D}) = (\mathcal{M}(\mathcal{D}))_{\mathsf{sat}}$ (hence, it is an H-complete RDF graph).
- $\mathcal{M}_{\mathsf{sat}}$ does not depend on the SPARQL query q, hence it can be pre-computed.
- It can be optimized (by exploiting query containment).

Mapping optimization – Exercise

Saturated mapping assertions $\mathcal{M}_{\text{sat}}$

$$
\begin{array}{llr}
\ldots & & \\
\texttt{academic}(acode, fn, ln, pos) & \rightsquigarrow \text{Faculty}(\textbf{iri2}(acode)) & (5)\\
\texttt{student}(scode, fn, ln) & \rightsquigarrow \text{Person}(\textbf{iri1}(scode)) & (6)\\
\texttt{academic}(acode, fn, ln, pos),\ pos = 9 & \rightsquigarrow \text{Faculty}(\textbf{iri2}(acode)) & (7)\\
\texttt{academic}(acode, fn, ln, pos),\ pos = 9 & \rightsquigarrow \text{Person}(\textbf{iri2}(acode)) & (8)\\
\texttt{academic}(acode, fn, ln, pos),\ pos = 2 & \rightsquigarrow \text{Faculty}(\textbf{iri2}(acode)) & (9)\\
\texttt{academic}(acode, fn, ln, pos),\ pos = 2 & \rightsquigarrow \text{Person}(\textbf{iri2}(acode)) & (10)\\
\texttt{academic}(acode, fn, ln, pos) & \rightsquigarrow \text{Person}(\textbf{iri2}(acode)) & (11)\\
\texttt{teaching}(course, acode) & \rightsquigarrow \text{Faculty}(\textbf{iri2}(acode)) & (12)\\
\texttt{teaching}(course, acode) & \rightsquigarrow \text{Person}(\textbf{iri2}(acode)) & (13)
\end{array}
$$

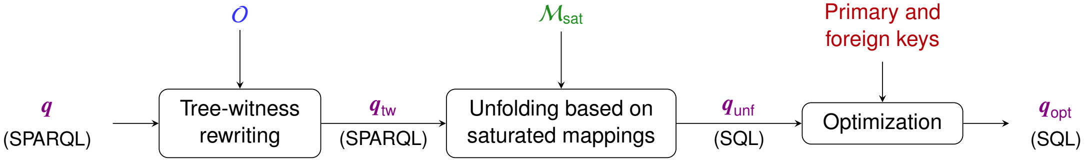
Consider also a foreign key over the database relations

FK: $\exists y_1.\texttt{teaching}(y_1, x) \rightarrow \exists y_2 y_3 y_4.\texttt{academic}(x, y_2, y_3, y_4)$

We can **optimize the mapping** using query containment and the FK. This removes mapping assertions 7, 8, 9, 10, 12, and 13.

$$
\begin{array}{llr}
\ldots & & \\
\texttt{academic}(acode, fn, ln, pos) & \rightsquigarrow \text{Faculty}(\textbf{iri2}(acode)) & (5)\\
\texttt{student}(scode, fn, ln) & \rightsquigarrow \text{Person}(\textbf{iri1}(scode)) & (6)\\
\texttt{academic}(acode, fn, ln, pos) & \rightsquigarrow \text{Person}(\textbf{iri2}(acode)) & (11)
\end{array}
$$

Query reformulation as implemented by the Ontop system



	Step	Input	Output
1.	Tree-witness rewriting	q and $\mathcal{O}$	q_{tw}
2.	Unfolding	q_{tw} and $\mathcal{M}_{\text{sat}}$	q_{unf}
3.	Optimization	q_{unf}, primary and foreign keys	q_{opt}

Let us now consider the optimization step.

Outline

1. Challenges in Data Access
2. Virtual Knowledge Graphs for Data Access (and Integration)
3. Optimizing Query Answering in VKGs
   - Query rewriting wrt an OWL 2 QL ontology
   - Query unfolding wrt a mapping
   - Mapping saturation
   - Optimization of query reformulation
4. Designing VKG Mappings
5. The Ontop System
6. Conclusions

SQL query optimization

Objective : produce SQL queries that are . . .

- similar to manually written ones
- adapted to existing query planners

Structural optimization

- From join-of-unions to union-of-joins
- IRI decomposition to improve performance of joins

Semantic optimization

- Redundant join elimination
- Redundant union elimination
- Using functional constraints

Integrity constraints

- Primary and foreign keys, uniqueness constraints
- Sometimes implicit
- Vital for query reformulation!

Reformulation example – 1. Unfolding

Saturated mapping

$\texttt{academic}(acode, fn, ln, pos),\ pos \in [1..8]$

$\rightsquigarrow \mathsf{Teacher}(\mathbf{iri2}(acode))$

$\texttt{teaching}(course, acode) \rightsquigarrow \mathsf{Teacher}(\mathbf{iri2}(acode))$

$\texttt{student}(scode, fn, ln) \rightsquigarrow \mathsf{firstName}(\mathbf{iri1}(scode), fn)$

$\texttt{academic}(acode, fn, ln, pos) \rightsquigarrow \mathsf{firstName}(\mathbf{iri2}(acode), fn)$

$\texttt{student}(scode, fn, ln) \rightsquigarrow \mathsf{lastName}(\mathbf{iri1}(scode), ln)$

$\texttt{academic}(acode, fn, ln, pos) \rightsquigarrow \mathsf{lastName}(\mathbf{iri2}(acode), ln)$

Query (we assume that the ontology is empty, hence $\boldsymbol{q}_{\mathsf{r}} = \boldsymbol{q}$)

$$\boldsymbol{q}(x, y, z) \leftarrow \mathsf{Teacher}(x),\ \mathsf{firstName}(x, y),\ \mathsf{lastName}(x, z)$$

We apply **query unfolding**, and then **normalization** to make the join conditions explicit.

$$\begin{aligned}
\boldsymbol{q}_{\mathsf{norm}}(x, y, z) &\leftarrow \boldsymbol{q}1_{\mathsf{unf}}(x),\ \boldsymbol{q}2_{\mathsf{unf}}(x_1, y),\\
&\ \ \boldsymbol{q}3_{\mathsf{unf}}(x_2, z),\ x = x_1,\ x = x_2\\
\boldsymbol{q}1_{\mathsf{unf}}(\mathbf{iri2}(acode)) &\leftarrow \texttt{academic}(acode, fn, ln, pos),\\
&\ \ pos \in [1..8]\\
\boldsymbol{q}1_{\mathsf{unf}}(\mathbf{iri2}(acode)) &\leftarrow \texttt{teaching}(course, acode)\\
\boldsymbol{q}2_{\mathsf{unf}}(\mathbf{iri1}(scode), fn) &\leftarrow \texttt{student}(scode, fn, ln)\\
\boldsymbol{q}2_{\mathsf{unf}}(\mathbf{iri2}(acode), fn) &\leftarrow \texttt{academic}(acode, fn, ln, pos)\\
\boldsymbol{q}3_{\mathsf{unf}}(\mathbf{iri1}(scode), ln) &\leftarrow \texttt{student}(scode, fn, ln)\\
\boldsymbol{q}3_{\mathsf{unf}}(\mathbf{iri2}(acode), ln) &\leftarrow \texttt{academic}(acode, fn, ln, pos)
\end{aligned}$$

Reformulation example – 2. Structural optimization

Unfolded normalized query

$$
\begin{aligned}
q_{\mathsf{norm}}(x,y,z) &\leftarrow q1_{\mathsf{unf}}(x),\ q2_{\mathsf{unf}}(x_1,y),\\
&\ q3_{\mathsf{unf}}(x_2,z),\\
&\ x = x_1,\ x = x_2\\
q1_{\mathsf{unf}}(\mathbf{iri2}(a)) &\leftarrow \texttt{academic}(a,f,l,p),\\
&\ p \in [1..8]\\
q1_{\mathsf{unf}}(\mathbf{iri2}(a)) &\leftarrow \texttt{teaching}(c,a)\\
q2_{\mathsf{unf}}(\mathbf{iri1}(s),f) &\leftarrow \texttt{student}(s,f,l)\\
q2_{\mathsf{unf}}(\mathbf{iri2}(a),f) &\leftarrow \texttt{academic}(a,f,l,p)\\
q3_{\mathsf{unf}}(\mathbf{iri1}(s),l) &\leftarrow \texttt{student}(s,f,l)\\
q3_{\mathsf{unf}}(\mathbf{iri2}(a),l) &\leftarrow \texttt{academic}(a,f,l,p)
\end{aligned}
$$

- While flattening, we can avoid to generate those queries that contain in their body an equality between two terms with incompatible IRI templates.
- This might avoid a potential exponential blowup.

Flattening (URI template lifting) – Part 1/2

$$
\begin{aligned}
q_{\mathsf{lift}}(\mathbf{iri2}(a),y,z) &\leftarrow \texttt{academic}(a,f_1,l_1,p_1),\\
&\ \texttt{student}(s,f_2,l_2),\\
&\ \texttt{student}(s_1,f_3,l_3),\\
&\ \mathbf{iri2}(a) = \mathbf{iri1}(s),\\
&\ \mathbf{iri2}(a) = \mathbf{iri1}(s_1),\\
&\ p_1 \in [1..8]
\end{aligned}
$$

$$
\begin{aligned}
q_{\mathsf{lift}}(\mathbf{iri2}(a),y,z) &\leftarrow \texttt{academic}(a,f_1,l_1,p_1),\\
&\ \texttt{student}(s,f_2,l_2),\\
&\ \texttt{academic}(a_2,f_3,z,p_3),\\
&\ \mathbf{iri2}(a) = \mathbf{iri1}(s),\\
&\ \mathbf{iri2}(a) = \mathbf{iri2}(a_2),\\
&\ p_1 \in [1..8]
\end{aligned}
$$

(One sub-query not shown)

$$
\begin{aligned}
q_{\mathsf{lift}}(\mathbf{iri2}(a),y,z) &\leftarrow \texttt{academic}(a,f_1,l_1,p_1),\\
&\ \texttt{academic}(a_1,y,l_2,p_2),\\
&\ \texttt{academic}(a_2,f_3,z,p_3),\\
&\ \mathbf{iri2}(a) = \mathbf{iri2}(a_1),\\
&\ \mathbf{iri2}(a) = \mathbf{iri2}(a_2),\\
&\ p_1 \in [1..8]
\end{aligned}
$$

Reformulation example – 2. Structural optimization

Unfolded normalized query

$$
\begin{aligned}
q_{\mathsf{norm}}(x,y,z) &\leftarrow q1_{\mathsf{unf}}(x),\ q2_{\mathsf{unf}}(x_1,y),\\
& q3_{\mathsf{unf}}(x_2,z),\\
& x = x_1,\ x = x_2\\
q1_{\mathsf{unf}}(\mathbf{iri2}(a)) &\leftarrow \mathtt{academic}(a,f,l,p),\\
& p \in [1..8]\\
q1_{\mathsf{unf}}(\mathbf{iri2}(a)) &\leftarrow \mathtt{teaching}(c,a)\\
q2_{\mathsf{unf}}(\mathbf{iri1}(s),f) &\leftarrow \mathtt{student}(s,f,l)\\
q2_{\mathsf{unf}}(\mathbf{iri2}(a),f) &\leftarrow \mathtt{academic}(a,f,l,p)\\
q3_{\mathsf{unf}}(\mathbf{iri1}(s),l) &\leftarrow \mathtt{student}(s,f,l)\\
q3_{\mathsf{unf}}(\mathbf{iri2}(a),l) &\leftarrow \mathtt{academic}(a,f,l,p)
\end{aligned}
$$

- While flattening, we can avoid to generate those queries that contain in their body an equality between two terms with incompatible IRI templates.
- This might avoid a potential exponential blowup.

Flattening (URI template lifting) – Part 2/2

$$
\begin{aligned}
q_{\mathsf{lift}}(\mathbf{iri2}(a),y,z) &\leftarrow \mathtt{teaching}(c,a),\\
& \mathtt{student}(s,f_2,l_2),\\
& \mathtt{student}(s_1,f_3,l_3),\\
& \mathbf{iri2}(a) = \mathbf{iri1}(s),\\
& \mathbf{iri2}(a) = \mathbf{iri1}(s_1)
\end{aligned}
$$

$$
\begin{aligned}
q_{\mathsf{lift}}(\mathbf{iri2}(a),y,z) &\leftarrow \mathtt{teaching}(c,a),\\
& \mathtt{student}(s,f_2,l_2),\\
& \mathtt{academic}(a_2,f_3,z,p_3),\\
& \mathbf{iri2}(a) = \mathbf{iri1}(s),\\
& \mathbf{iri2}(a) = \mathbf{iri2}(a_2)
\end{aligned}
$$

(One sub-query not shown)

$$
\begin{aligned}
q_{\mathsf{lift}}(\mathbf{iri2}(a),y,z) &\leftarrow \mathtt{teaching}(c,a),\\
& \mathtt{academic}(a_1,y,l_2,p_2),\\
& \mathtt{academic}(a_2,f_3,z,p_3),\\
& \mathbf{iri2}(a) = \mathbf{iri2}(a_1),\\
& \mathbf{iri2}(a) = \mathbf{iri2}(a_2)
\end{aligned}
$$

Reformulation example – 3. Semantic optimization

We are left with just two queries, which we can simplify by eliminating equalities

$$
\begin{array}{lll}
q_{\text{struct}}(\textbf{iri2}(a), y, z) & \leftarrow & \texttt{academic}(a, f_1, l_1, p_1),\ p_1 \in [1..8], \\
& & \texttt{academic}(a, y, l_2, p_2), \\
& & \texttt{academic}(a, f_3, z, p_3) \\
q_{\text{struct}}(\textbf{iri2}(a), y, z) & \leftarrow & \texttt{teaching}(c, a), \\
& & \texttt{academic}(a, y, l_2, p_2), \\
& & \texttt{academic}(a, f_3, z, p_3)
\end{array}
$$

We can then exploit database constraints (e.g., primary keys) for semantic optimization of the query.

Self-join elimination (semantic optimization)

PK: $\texttt{academic}(acode, f, l, p) \land \texttt{academic}(acode, f', l', p') \rightarrow (f = f') \land (l = l') \land (p = p')$

$$
\begin{array}{lll}
q_{\text{opt}}(\textbf{iri2}(a), y, z) & \leftarrow & \texttt{academic}(a, y, z, p_1),\ p_1 \in [1..8] \\
q_{\text{opt}}(\textbf{iri2}(a), y, z) & \leftarrow & \texttt{teaching}(c, a),\ \texttt{academic}(a, y, z, p_2)
\end{array}
$$

Outline

unibz

Designing VKG mappings

The form of the mapping 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 can rely on some **simple observations**:

- 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 mapping (and ontology axioms), is signaled by the presence of some (combination of) constraints that hold 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.

VKG mapping patterns

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.

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

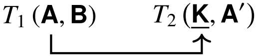
- Consider also constraints emerging from specific configurations of the data in the DB.
- For each schema-driven pattern, we identify a data-driven version:
  The constraints over the schema are not explicitly specified, but hold in the data.
- We provide also data-driven patterns that do not have a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

- We use also additional semantic information from the ontology ⇝ **Pattern modifiers**
- 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 traditional types of DB constraints:

- **Primary key constraint**: $T(\underline{\mathbf{K}}, \mathbf{A})$
- **Key constraint**: $\mathit{key}_T(\mathbf{K})$
- **Foreign key constraint**: $T_1[\mathbf{A}] \subseteq T_2[\mathbf{K}]$, where **K** is a (typically primary) key of relation T_2.
  We use the notation:



$$T_1\,(\mathbf{A}, \mathbf{B}) \qquad T_2\,(\underline{\mathbf{K}}, \mathbf{A}')$$

Note: We use normal font (e.g., A) for single attributes, and boldface for sets of attributes (e.g., **A**).

Types of mapping patterns

In the following, we discuss the following mapping patterns:

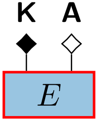
- Entity (MpE)
- Relationship (MpR)
- Relationship with Identifier Alignment (MpRa)
- Relationship with Merging (MpRm)
- 1-1 Relationship with Merging (MpR11m)
- Reified Relationship (MpRR)
- Hierarchy (MpH)
- Hierarchy with Identifier Alignment (MpHa)
- Clustering Entity to Class / Data Property / Object Property (MpCE2X)

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 source and target part of the resulting mapping assertion(s).
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:
$\{ \exists d_A \sqsubseteq C_E \}_{A \in \mathbf{K} \cup \mathbf{A}}$

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

- This fundamental 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) – Example

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

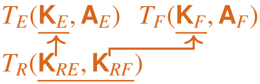
`TClient(ssn,name,dateOfBirth,hobbies)`

Mapping: TClient is mapped to a Client class using the attributes ssn to construct its objects. In addition, the ssn, name, and dateOfBirth 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} .
```

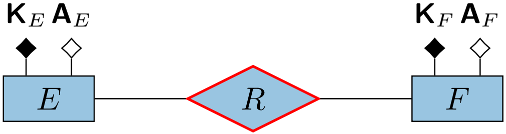
Mapping pattern: Relationship (MpR)

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 Person and the range an additional class Location, corresponding to the TLocation table.

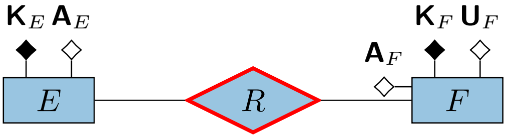
```
mappingId MAddress
source    SELECT client, locCity, locStreet FROM TAddress
target    :C/{client} :address :L/{locCity}/{locStreet} .
```

Mapping pattern: Relationship with Identifier Alignment (MpRa)

Relational schema and constraints:

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

$T_R(\underline{\mathbf{K}_{RE}, \mathbf{U}_{RF}})$ $\mathit{key}_{R_F}(\mathbf{U}_F)$



Mapping assertion:

$s : T_R \bowtie_{\mathbf{U}_{RF}=\mathbf{U}_F} T_F$

$t :$ `: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 $\mathbf{U}_F$.

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)    key[TLocation]: city,street
TAddress(client,locCity,locStreet)
    FK: TAddress[client] -> Tclient[ssn]
    FK: TAddress[locCity,locStreet] -> TLocationCoord[city,street]
```

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

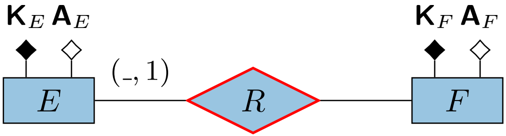
```
mappingId MAddressCoord
source    SELECT client, latitude, longitude
          FROM TAddress JOIN TLocationCoord ON locCity = city AND locStreet = street
target    :C/{client} :address :LC/{latitude}/{longitude} .
```

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_R `:F/{`$\mathbf{K}_{EF}$`}` .

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 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 $\mathbf{K}_{EF}$.

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.

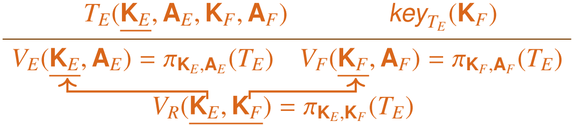
```
TLocation(city,street)
TClient(ssn,name,dateOfBirth,billCity,billStreet,hobbies)
    FK: TClient[billCity,billStreet] -> TLocation[city,street]
```

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)

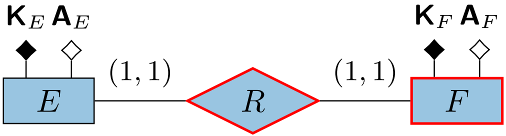
Relational schema and constraints:



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

$V_E(\underline{\mathbf{K}_E}, \mathbf{A}_E) = \pi_{\mathbf{K}_E,\mathbf{A}_E}(T_E)$ $\qquad$ $V_F(\underline{\mathbf{K}_F}, \mathbf{A}_F) = \pi_{\mathbf{K}_F,\mathbf{A}_F}(T_E)$

$V_R(\underline{\mathbf{K}_E, \mathbf{K}_F}) = \pi_{\mathbf{K}_E,\mathbf{K}_F}(T_E)$



Mapping assertion:

$s : T_E$

t : $\texttt{:F/\{}\mathbf{K}_F\texttt{\} rdf:type}\ C_F$.

$\{\ \texttt{:F/\{}\mathbf{K}_F\texttt{\}}\ d_A\ \ \{A\}\ .\ \}_{A\in\mathbf{K}_F\cup\mathbf{A}_F}$

$\texttt{:E/\{}\mathbf{K}_E\texttt{\}}\ p_R\ \ \texttt{:F/\{}\mathbf{K}_F\texttt{\}}$.

Ontology axioms:

$\{\exists d_A \sqsubseteq C_F\}_{A\in\mathbf{K}_F\cup\mathbf{A}_F}$

$\exists p_R \sqsubseteq C_E$

$\exists p_R^- \sqsubseteq C_F$

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 $\texttt{:F/\{}\mathbf{K}_F\texttt{\}}$ 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 **SR11m** 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)
    key[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.

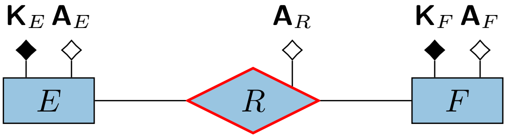
  Distinguishing which of these two possibilities is the correct one, requires in-depth knowledge about the domain.
- The necessary domain knowledge may also come from the ontology, e.g., if the data properties corresponding to the attributes are already present in the ontology, and their domain has been declared.

Mapping pattern: Reified Relationship (MpRR) – Attribute case

Relational schema and constraints:

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

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



Mapping assertion:

$s : T_R$

$t :\ \texttt{:R/\{}\mathbf{K}_{RE}\texttt{\}/\{}\mathbf{K}_{RF}\texttt{\}}\ \texttt{rdf:type}\ C_R\ .$

$\{\ \texttt{:R/\{}\mathbf{K}_{RE}\texttt{\}/\{}\mathbf{K}_{RF}\texttt{\}}\ d_A\ \{A\}\ .\ \}_{A\in\mathbf{K}_{RE}\cup\mathbf{K}_{RF}\cup\mathbf{A}_R}$

$\texttt{:R/\{}\mathbf{K}_{RE}\texttt{\}/\{}\mathbf{K}_{RF}\texttt{\}}\ p_{RE}\ \texttt{:E/\{}\mathbf{K}_{RE}\texttt{\}}\ .$

$\texttt{:R/\{}\mathbf{K}_{RE}\texttt{\}/\{}\mathbf{K}_{RF}\texttt{\}}\ p_{RF}\ \texttt{:F/\{}\mathbf{K}_{RF}\texttt{\}}\ .$

Ontology axioms:

$\exists p_{RE} \sqsubseteq C_R$ $\exists p_{RF} \sqsubseteq C_R$

$\exists p_{RE}^- \sqsubseteq C_E$ $\exists p_{RF}^- \sqsubseteq C_F$

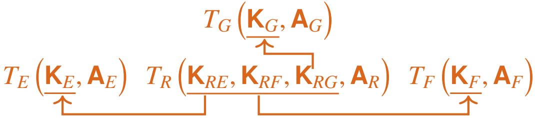
$\{\ \exists d_A^- \sqsubseteq C_R\ \}_{A\in\mathbf{K}_{RE}\cup\mathbf{K}_{RF}\cup\mathbf{A}}$

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 $\texttt{:R/\{}\mathbf{K}_{RE}\texttt{\}/\{}\mathbf{K}_{RF}\texttt{\}}$.
- The mapping ensures that each components 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.

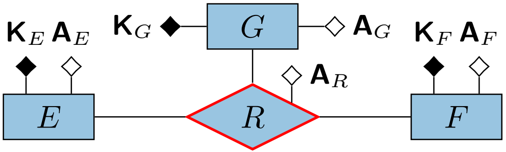
Mapping pattern: Reified Relationship (MpRR) – n-ary relationship case

Relational schema and constraints:



$T_G(\underline{\mathbf{K}_G}, \mathbf{A}_G)$

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E)$ $T_R(\underline{\mathbf{K}_{RE}, \mathbf{K}_{RF}, \mathbf{K}_{RG}}, \mathbf{A}_R)$ $T_F(\underline{\mathbf{K}_F}, \mathbf{A}_F)$



Mapping assertion: $\mathbf{K}_R := \mathbf{K}_{RE} \cup \mathbf{K}_{RF} \cup \mathbf{K}_{RG}$

$s : T_R$

$t : \texttt{:R/\{}\mathbf{K}_R\texttt{\}}\ \texttt{rdf:type}\ C_R\ .$

$\quad \{\ \texttt{:R/\{}\mathbf{K}_R\texttt{\}}\ d_A\ \ \{A\}\ .\ \}_{A \in \mathbf{K}_R \cup \mathbf{A}_R}$

$\quad \texttt{:R/\{}\mathbf{K}_R\texttt{\}}\ p_{RE}\ \ \texttt{:E/\{}\mathbf{K}_{RE}\texttt{\}}\ .$

$\quad \texttt{:R/\{}\mathbf{K}_R\texttt{\}}\ p_{RF}\ \ \texttt{:F/\{}\mathbf{K}_{RF}\texttt{\}}\ .\quad \texttt{:R/\{}\mathbf{K}_R\texttt{\}}\ p_{RG}\ \ \texttt{:G/\{}\mathbf{K}_{RG}\texttt{\}}\ .$

Ontology axioms:

$\exists p_{RE} \sqsubseteq C_R \qquad \exists p_{RF} \sqsubseteq C_R \qquad \exists p_{RG} \sqsubseteq C_R$

$\exists p_{RE}^- \sqsubseteq C_E \qquad \exists p_{RF}^- \sqsubseteq C_F \qquad \exists p_{RG}^- \sqsubseteq C_G$

$\{\ \exists d_A^- \sqsubseteq C_R\ \}_{A \in \mathbf{K}_R \cup \mathbf{A}_R}$

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)                            FK: TExam[student] -> TStudent[ssn]
TCourse(cid,cname,credits)                     FK: TExam[course] -> TCourse[cid]
TProfessor(ssn,pname,level)                    FK: TExam[professor] -> TProfessor[ssn]
```

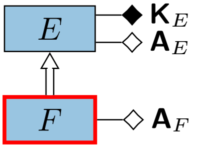
Mapping: This information is represented by a relationship that is inherently ternary. The ontology should contain a class Exam corresponding to the reified relationship, connected via three object properties to the classes Student, Course, and Professor. The mapping ensures that the class Exam is instantiated 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:

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



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:

$C_F \sqsubseteq C_E$

$\{\ \exists d_A^- \sqsubseteq C_F\ \}_{A \in \mathbf{A}_F}$

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,dateOfBirth)
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} ;
                   :dob {dateOfBirth} .

mappingId MStudent
source    SELECT ssn, sid FROM TStudent
target    :P/{ssn} rdf:type :Student ;
                   :studentId {sid} ;
                   :credits {credits} .
```

Mapping pattern: Hierarchy with Identifier Alignment (MpHa)

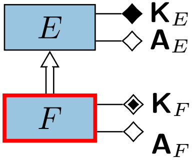
Relational schema and constraints:

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

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

$T_E(\underline{\mathbf{K}_E}, \mathbf{A}_E) \quad key_{V_F}(\mathbf{K}_F)$

$V_F(\mathbf{K}_F, \underline{\mathbf{U}_F}, \mathbf{A}_F) = T_F$



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:

$C_F \sqsubseteq C_E$

$\{ \exists d_A^- \sqsubseteq C_F \}_{A \in \mathbf{K}_F \cup \mathbf{A}_F}$

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,dateOfBirth)
TStudent(sid,ssn,credits)        FK: TStudent[ssn] -> TPerson[ssn]      key[TStudent]: ssn
TEnrolled(student,course)        FK: TEnrolled[student] -> TStudent[sid]
```

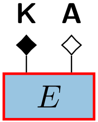
Mapping: By applying pattern **MpHa**, we identify the instances of Student by their ssn, and we create a view `VStudent(sid,ssn,credits)`. But now, considering this view instead of TStudent, in order to map TEnrolled into an object property enrolledIn, we need to apply pattern **MpRa** rather than **MpR**.

```
mappingId MPerson
source    SELECT ssn, name, dob FROM TPerson
target    :P/{ssn} rdf:type :Person ;    :name {name} ;    :dob {dateOfBirth} .

mappingId MStudent
source    SELECT sid, ssn, credits FROM TStudent
target    :P/{ssn} rdf:type :Student ;   :studentId {sid} ;   :credits {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)

Relational schema and constraints:
$T_E(\underline{\mathbf{K}}, \mathbf{A})$,
$\mathbf{B} \subseteq \mathbf{K} \cup \mathbf{A}$ such that $\mathit{partition}_{\mathcal{D}}(\mathbf{B}, T_E)$

$\{\ V_{E_{\mathbf{v}}}(\mathbf{K}, \mathbf{A}) = \sigma_{\mathbf{B}=\mathbf{v}}(T_E)\ \}_{\mathbf{v} \in \pi_{\mathbf{B}}(T_E)}$



$\mathbf{B} \subseteq \mathbf{K} \cup \mathbf{A}$
such that $\mathit{partition}_{\mathcal{D}}(\mathbf{B}, E)$

Mapping assertions:
$\{\ s : \sigma_{\mathbf{B}=\mathbf{v}} T_E$
t : `:E/{`$\mathbf{K}$`} 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 TPerson containing persons with an attribute defining their gender and ranging over '`F`' or '`M`'.

`TPerson(ssn,name,dob,gender)`

Mapping: The ontology defines a class Person with two subclasses Female and Male. Pattern **MpCE2C** clusters the table according to the gender attribute, and instantiates the classes Female and Male accordingly.

```
mappingId MPerson
source    SELECT ssn, name, dob FROM TPerson
target    :P/{ssn} rdf:type :Person ;    :name {name} ;    :dob {dateOfBirth} .

mappingId MFemale
source    SELECT ssn FROM TPerson WHERE gender = 'F'
target    :P/{ssn} rdf:type :Female .

mappingId MMale
source    SELECT ssn FROM TPerson WHERE gender = 'M'
target    :P/{ssn} rdf:type :Male .
```

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 (as we have seen).

- 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.

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.

Outline

The *Ontop* system [C., Cogrel, et al. 2017, Semantic Web J.], [Xiao, Lanti, et al. 2020, ISWC]

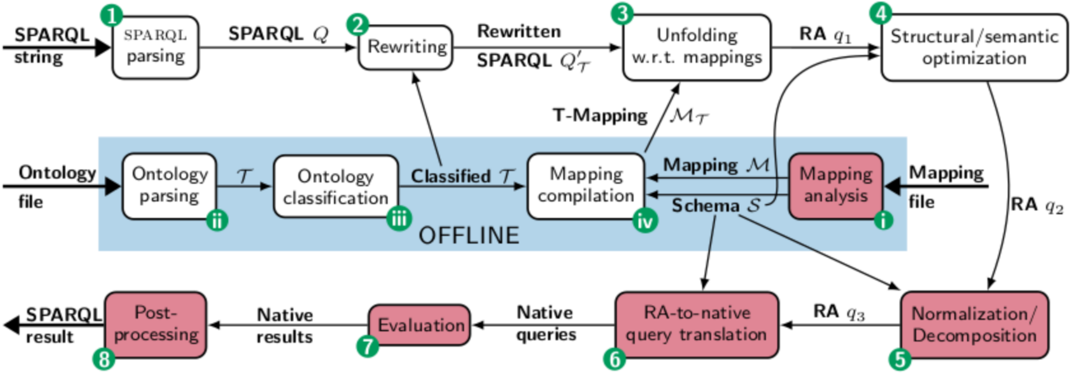
ontop

`https://ontop-vkg.org/`

- State-of-the-art VKG system.
- Implements the presented techniques for query answering and optimization.
- Addresses the key challenges of scalability and performance.
- Compliant with all relevant Semantic Web standards:
  RDF, RDFS, OWL 2 QL, R2RML, SPARQL, and GeoSPARQL.
- Supports all major relational DBMSs:
  Oracle, DB2, MS SQL Server, Postgres, MySQL, Teiid, Dremio, Denodo, etc.
- Open-source and released under Apache 2 license.

unibz

Query answering in *Ontop*



The *Ontopic* spinoff of unibz

ONTOPIC

https://ontopic.ai/

Funded in April 2019 as the first spin-off of the Free University of Bozen-Bolzano.

- Ontopic Studio ready to be released
  - Ensures scalability, reliability, and cost-efficiency at design and runtime of VKG solutions.
  - Strong focus on usability.
- Technical services
  - Technical support for Ontop and Ontopic Studio.
  - Customized developments.
- Consulting on adoption of VKG-based solutions for data access and integration.

unibz

Outline

1. Challenges in Data Access
2. Virtual Knowledge Graphs for Data Access (and Integration)
3. Optimizing Query Answering in VKGs
4. Designing VKG Mappings
5. The Ontop System
6. Conclusions

Conclusions

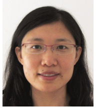
- VKGs are by now a mature technology to address the challenges related to data access and integration.
- It has been well-investigated and applied in many different scenarios mostly for the case of relational data sources.
- The technology is general purpose, and it can be tailored towards specific domains, relying also on standard ontologies.
- Performance and scalability w.r.t. larger datasets (volume), larger and more complex ontologies (variety, veracity), and multiple heterogeneous data sources (variety, volume) is a challenge.
- Currently, VKGs are being investigated for alternative types of data, such as **temporal data**, **graph data**, **tree structured data**, **linked open data**, and **geo-spatial data**.
- Performance and scalability are even more critical for these more complex domains.

Thank you!

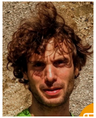
A great thank you to all my collaborators


Elena Botoeva

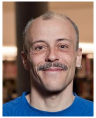

Julien Corman


Linfang Ding


Elem Güzel


Davide Lanti


Marco Montali

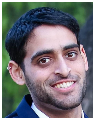

Alessandro Mosca


Mariano Rodriguez Muro


Guohui Xiao

Technion Haifa


Avigdor Gal


Roee Shraga

Birkbeck College London

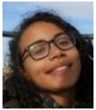

Roman Kontchakov


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

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] 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. DOI: `10.1007/s10817-007-9078-x`.

[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. DOI: `10.1016/j.datak.2011.09.004`.

References III

[11] 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`.

[12] Alessandro Artale, Diego C., Roman Kontchakov & Michael Zakharyaschev. *The DL-Lite Family and Relations*. Tech. rep. BBKCS-09-03. Available at `http://www.dcs.bbk.ac.uk/research/techreps/2009/bbkcs-09-03.pdf`. London: School of Computer Science and Information Systems, Birbeck College, 2009.

[13] Diego C., Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini & Riccardo Rosati. "Data Complexity of Query Answering in Description Logics". In: *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*. 2006, pp. 260–270.

[14] Diego C., Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini & Riccardo Rosati. "Data Complexity of Query Answering in Description Logics". In: *Artificial Intelligence* 195 (2013), pp. 335–360. DOI: `10.1016/j.artint.2012.10.003`.

References IV

[15] Stanislav Kikot, Roman Kontchakov & Michael Zakharyaschev. “Conjunctive Query Answering with OWL 2 QL”. In: *Proc. of the 13th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*. 2012, pp. 275–285.

[16] Mariano Rodriguez-Muro, Roman Kontchakov & Michael Zakharyaschev. “Ontology-Based Data Access: Ontop of Databases”. In: *Proc. of the 12th Int. Semantic Web Conf. (ISWC)*. Vol. 8218. Lecture Notes in Computer Science. Springer, 2013, pp. 558–573. DOI: `10.1007/978-3-642-41335-3_35`.

[17] Roman Kontchakov, Martin Rezk, Mariano Rodriguez-Muro, Guohui Xiao & Michael Zakharyaschev. “Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime”. In: *Proc. of the 13th Int. Semantic Web Conf. (ISWC)*. Vol. 8796. Lecture Notes in Computer Science. Springer, 2014, pp. 552–567. DOI: `10.1007/978-3-319-11964-9_35`.

unibz

References V

[18] Roman Kontchakov & Michael Zakharyaschev. “An Introduction to Description Logics and Query Rewriting”. In: *Reasoning Web: Reasoning on the Web in the Big Data Era – 10th Int. Summer School Tutorial Lectures (RW)*. Vol. 8714. Lecture Notes in Computer Science. Springer, 2014, pp. 195–244. DOI: `10.1007/978-3-319-10587-1_5`.

[19] 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.

[20] Diego C., Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro & Guohui Xiao. “Ontop: Answering SPARQL Queries over Relational Databases”. In: *Semantic Web J.* 8.3 (2017), pp. 471–487. DOI: `10.3233/SW-160217`.

[21] Guohui Xiao, Davide Lanti, Roman Kontchakov, Sarah Komla-Ebri, Elem Güzel-Kalayci, Linfang Ding, Julien Corman, Benjamin Cogrel, Diego C. & Elena Botoeva. “The Virtual Knowledge Graph System Ontop”. In: *Proc. of the 19th Int. Semantic Web Conf. (ISWC)*. Vol. 12507. Lecture Notes in Computer Science. Springer, 2020, pp. 259–277. DOI: `10.1007/978-3-030-62466-8_17`.