Designing Virtual Knowledge Graphs with Ontop and Ontopic Studio

Diego Calvanese[1,2], Benjamin Cogrel[1,2], Davide Lanti[2]
[1] Ontopic s.r.l.
[2] Free University of Bozen-Bolzano

15th Seminar on Ontology Research in Brazil (OntoBras 2022),

25 November 2022

Outline of the tutorial

1. Introduction to Virtual Knowledge Graphs (VKGs)
   45 min – Diego Calvanese

2. Introduction to Ontopic Studio
   45 min – Benjamin Cogrel

3. VKG Design with Ontopic Studio (handson)
   60 min – Benjamin Cogrel

4. Setting up and accessing a SPARQL endpoint with Ontop (handson)
   30 min – Davide Lanti

Part I

Introduction to Virtual Knowledge Graphs

Outline of Part 1

1. Challenges in Data Access

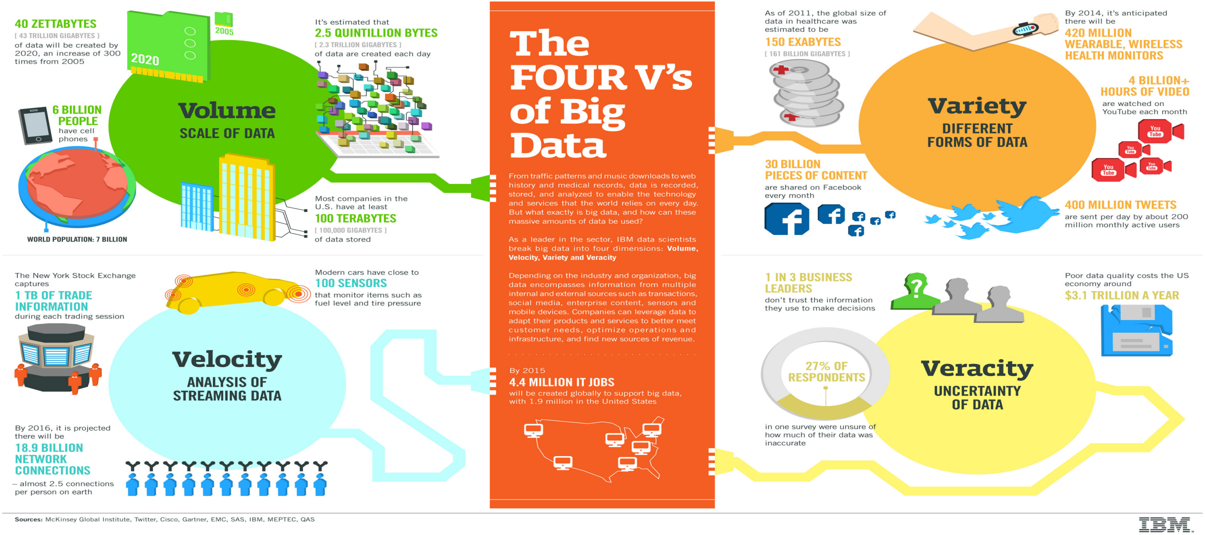
2. A Quick History of VKGs

3. Ontop

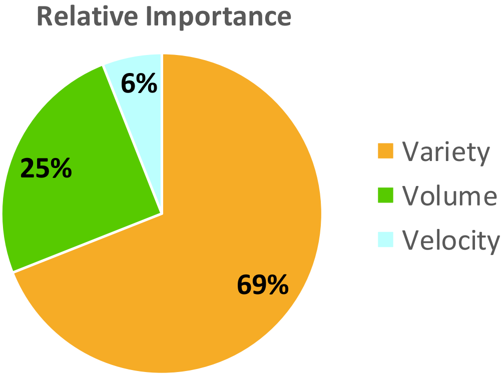
4. The VKG Framework

5. Query Answering in VKGs

Challenges in data management



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/

The problem of data access

In large organization data management is a complex challenge:

- Many different data sets are created independently.
- The data is heterogeneous in the way it is represented and structured.
- Data are often stored across different sources (possibly controlled by different people / organizations).

The problem of data access

In large organization data management is a complex challenge:

- Many different data sets are created independently.
- The data is heterogeneous in the way it is represented and structured.
- Data are often stored across different sources (possibly controlled by different people / organizations).

However, complex data processing pipelines (e.g., for analysis, monitoring and prediction) require to **access in an integrated and uniform way** such large, richly structured, and heterogeneous data sets.

Why heterogeneity?

- Data model heterogeneity: Relational data, graph data, xml, json, csv, text files, . . .
- System heterogeneity: Even when systems adopt the same data model, they are not always fully compatible.
- Schema heterogeneity: Different people see things differently, and design schemas differently!
- Data-level heterogeneity: e.g., ‘IBM’ vs. ‘Int. Business Machines’ vs. ‘International Business Machines’

Schema heterogeneity

Source 1

Movie (mid, title)
Actor (aid, firstName, lastName,
nationality, yearOfBirth)
Plays (aid, mid)
MovieDetails (mid, director, genre, year)

Source 2

Cinema (place, movie, start)

Source 3

NYCCinema (name, title, startTime)

Source 4

MovieGenre (title, genre)
MovieDirector (title, dir)
MovieYear (title, year)

Source 5

Review (title, date, grade, review)

Source 6

Movie (title, director, year, genre)
Actor (title, name)
Plays (movie, location, startTime)
Review (title, rating, description)

Schema heterogeneity

Organization of tables and attributes

Source 1
Movie (mid, title)
Actor (aid, firstName, lastName, nationality, yearOfBirth)
Plays (aid, mid)
MovieDetails (mid, director, genre, year)

Source 2
Cinema (place, movie, start)

Source 3
NYCCinema (name, title, startTime)

Source 4
MovieGenre (title, genre)
MovieDirector (title, dir)
MovieYear (title, year)

Source 5
Review (title, date, grade, review)

Source 6
Movie (title, director, year, genre)
Actor (title, name)
Plays (movie, location, startTime)
Review (title, rating, description)

Schema heterogeneity

Table and attribute names

Source 1

Movie (mid, title)
Actor (aid, firstName, lastName, nationality, yearOfBirth)
Plays (aid, mid)
MovieDetails (mid, director, genre, year)

Source 2

Cinema (place, movie, start)

Source 3

NYCCinema (name, title, startTime)

Source 4

MovieGenre (title, genre)
MovieDirector (title, dir)
MovieYear (title, year)

Source 5

Review (title, date, grade, review)

Source 6

Movie (title, director, year, genre)
Actor (title, name)
Plays (movie, location, startTime)
Review (title, rating, description)

Schema heterogeneity

Table and attribute names

Source 1

Movie (mid, title)
Actor (aid, firstName, lastName, nationality, yearOfBirth)
Plays (aid, mid)
MovieDetails (mid, director, genre, year)

Source 2

Cinema (place, movie, start)

Source 3

NYCCinema (name, title, startTime)

Source 4

MovieGenre (title, genre)
MovieDirector (title, dir)
MovieYear (title, year)

Source 5

Review (title, date, grade, review)

Source 6

Movie (title, director, year, genre)
Actor (title, name)
Plays (movie, location, startTime)
Review (title, rating, description)

Schema heterogeneity

Table and attribute names

Source 1

Movie (mid, title)
Actor (aid, firstName, lastName, nationality, yearOfBirth)
Plays (aid, mid)
MovieDetails (mid, director, genre, year)

Source 2

Cinema (place, movie, start)

Source 3

NYCCinema (name, title, startTime)

Source 4

MovieGenre (title, genre)
MovieDirector (title, dir)
MovieYear (title, year)

Source 5

Review (title, date, grade, review)

Source 6

Movie (title, director, year, genre)
Actor (title, name)
Plays (movie, location, startTime)
Review (title, rating, description)

Schema heterogeneity

Coverage and detail of the schema

Source 1

Movie (mid, title)
Actor (aid, firstName, lastName, nationality, yearOfBirth)
Plays (aid, mid)
MovieDetails (mid, director, genre, year)

Source 2

Cinema (place, movie, start)

Source 3

NYCCinema (name, title, startTime)

Source 4

MovieGenre (title, genre)
MovieDirector (title, dir)
MovieYear (title, year)

Source 5

Review (title, date, grade, review)

Source 6

Movie (title, director, year, genre)
Actor (title, name)
Plays (movie, location, startTime)
Review (title, rating, description)

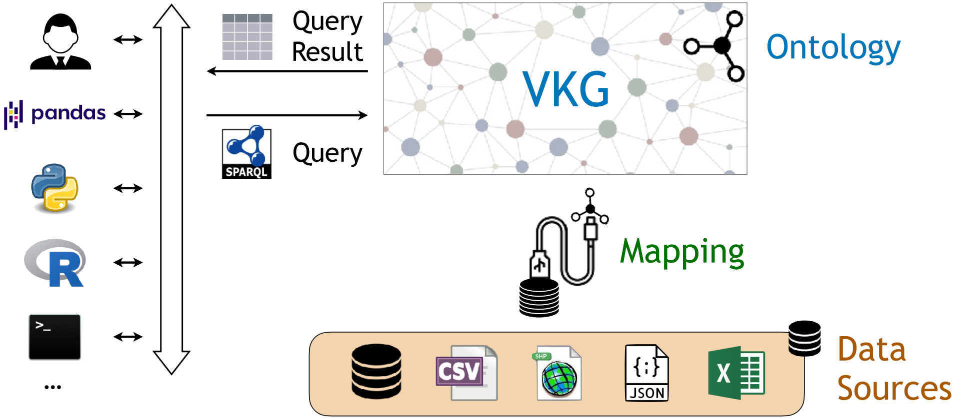
How can we address the complexity of data access?

We combine three key ideas:

1. Expose to users/applications the data in a very flexible data model, making use of terms the users are familiar with
   $\leadsto$ Knowledge Graph with vocabulary expressed in a domain ontology.
2. Map the data sources to the domain ontology to provide data for the KG.
3. Exploit virtualization, i.e., the KG is not materialized, but kept virtual.

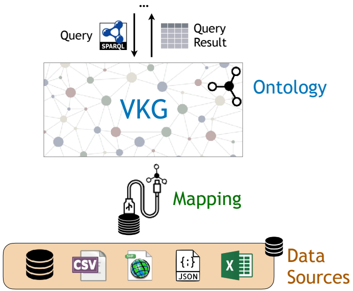
This gives rise to the Virtual Knowledge Graph (VKG) approach to data access, also called Ontology-based Data Access (OBDA).
[Xiao, Calvanese, et al. 2018, IJCAI]

Virtual Knowledge Graph (VKG) architecture



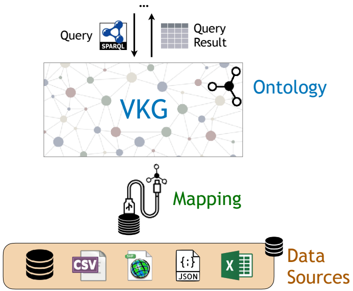
Why an ontology?

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



Why an ontology?

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

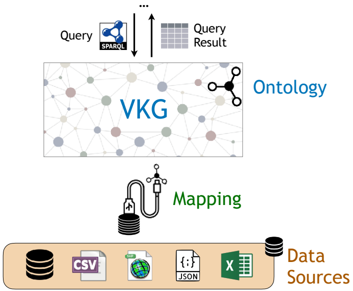


- In the VKG setting, the ontology has two purposes:
  - It defines a vocabulary of terms to denote classes and properties that are familiar to the user.
  - It extends the data in the sources with background knowledge about the domain of interest, and this knowledge is machine processable.
- One can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

Why a KG for the global schema?

Traditional approaches to data management rely on the relational model.

Why a KG for the global schema?



Traditional approaches to data management rely on the relational model.

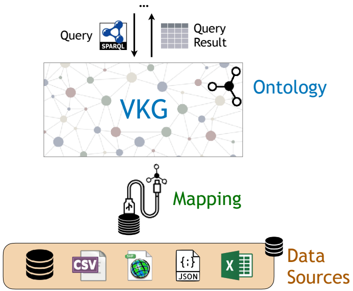
A Knowledge Graph, instead:

- 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 changes or new information.

Why mappings?

Traditional approaches to data access/integration rely on mediators, which are specified through complex code.

Why mappings?



Traditional approaches to data access/integration rely on mediators, which are specified through complex code.

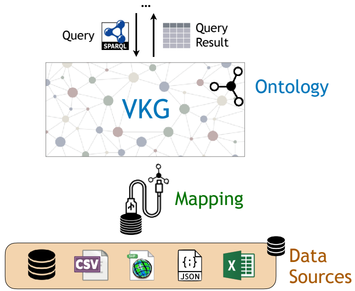
Mappings, instead:

- 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 access /1 integration relies on extract-transform-load (ETL) operations, to load data into an integrated data store / data warehouse / materialized KG.

Why virtualization?



Materialized data access /1 integration relies on extract-transform-load (ETL) operations, to load data into an integrated data store / data warehouse / materialized KG.

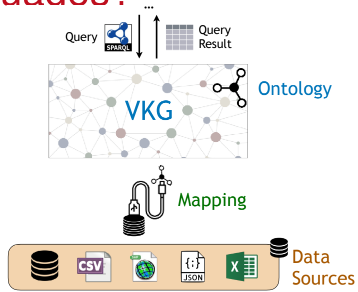
In the virtual approach, instead:

- 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 existing data infrastructure and expertise.

Engineering a VKG solution – Which languages?

Which are the right languages for the components of the VKG framework?

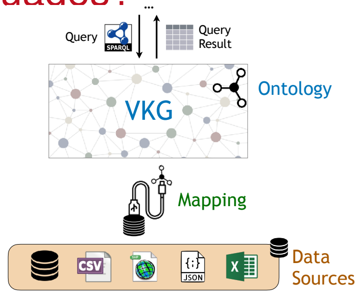
We need to consider the tradeoff between expressive power and efficiency, where efficiency with respect to the data is the key aspect to consider.



Engineering a VKG solution – Which languages?

Which are the right languages for the components of the VKG framework?

We need to consider the tradeoff between expressive power and efficiency, where efficiency with respect to the data is the key aspect to consider.



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

1. Challenges in Data Access

2. A Quick History of VKGs

3. Ontop

4. The VKG Framework

5. Query Answering in VKGs

Outline

1. Challenges in Data Access

2. A Quick History of VKGs

3. Ontop

4. The VKG Framework

5. Query Answering in VKGs

A quick history of VKGs

1990's Logic-based knowledge representation languages proposed as global schema formalisms in data integration: high expressive power, too complex $\rightsquigarrow$ mostly theoretical

2005 Families of lightweight ontology languages (or Description Logics) $\rightsquigarrow$ DL-Lite family of DLs

2007 DL-Lite used as a basis for the Ontology-based Data Access (OBDA) paradigm: based on conjunctive queries, abstract mapping language

2012 OWL 2 standardized by W3C with 3 profiles: OWL 2 QL profile based on DL-Lite

2012 R2RML mapping language standardized by W3C

> 2012 OBDA paradigm moved to Semantic Web standards

2019 OBDAs rebranded as VKGs

Outline

1. Challenges in Data Access
2. A Quick History of VKGs
3. Ontop
4. The VKG Framework
5. Query Answering in VKGs

The *Ontop* system

ontop

https://ontop-vkg.org/

- State-of-the-art VKG system
- Compliant with all relevant Semantic Web standards:
  RDF, RDFS, OWL 2 QL, R2RML, SPARQL, and GeoSPARQL
- Supports all major relational DBs:
  Oracle, DB2, MS SQL Server, Postgres, MySQL, Teiid, Dremio, Denodo, etc.
- Open-source and released under Apache 2 license.

Developer community

unibz

ONTOPIC

Birkbeck UNIVERSITY OF LONDON

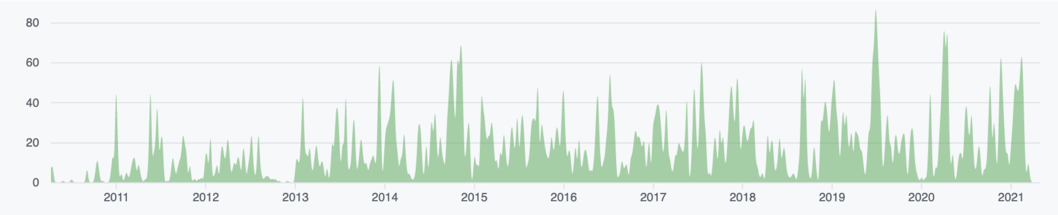
UiO : University of Oslo

HELLENIC REPUBLIC
National and Kapodistrian University of Athens

UNIVERSITÄT LEIPZIG

ontotext

POLITECNICO MILANO 1863



Some use cases of *Ontop* – Research projects

- EU FP7 project Optique "Scalable End-user Access to Big Data" (11/2012 – 10/2016)
  - 10 partners, including industrial partners Statoil, Siemens, DNV
  - *Ontop* is core component of the Optique platform
- EU project EPNet (ERC Advanced Grant) "Production and distribution of food during the Roman Empire: Economics and Political Dynamics"
  - Access to data in the cultural heritage domain [Calvanese et al. 2016, EAAI]
- Euregio project KAOS "Knowledge-aware Operational Support" (06/2016 – 05/2019)
  - Preparation of standardized log files from timestamped log data for the purpose of process mining
- EU H2020 project INODE "Intelligent Open Data Exploration" (11/2019 – 04/2023)
  - Development of techniques for the flexible interaction with data

See also [Xiao, Ding, et al. 2019].

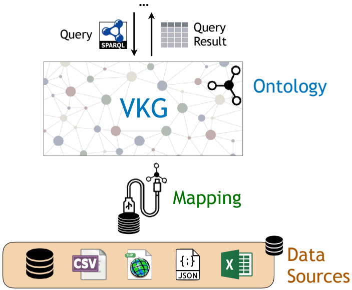
Some use cases of *Ontop* – Industrial applications

- Industry 4.0
  - Many vendors / historical data of exploration campaigns
  - Examples: Equinor, Siemens, Bosch
- Analytical / BI
  - Combine internal data, manual processes (e.g., Excel) and external data
  - Data privacy issues / GDPR: we need to avoid data copies
  - Examples: Toscana Open Research, a large European university
- Geospatial data
  - GeoSPARQL over PostGIS
  - Examples: LinkedGeoData.org, South Tyrolean Open Data Hub

Outline

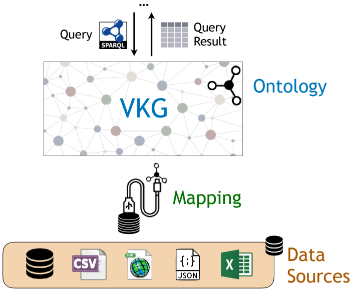
1. Challenges in Data Access
2. A Quick History of VKGs
3. Ontop
4. The VKG Framework
5. Query Answering in VKGs

Components of the VKG architecture



We consider now the main components that make up a VKG system, and the languages used to define them.

Components of the VKG architecture



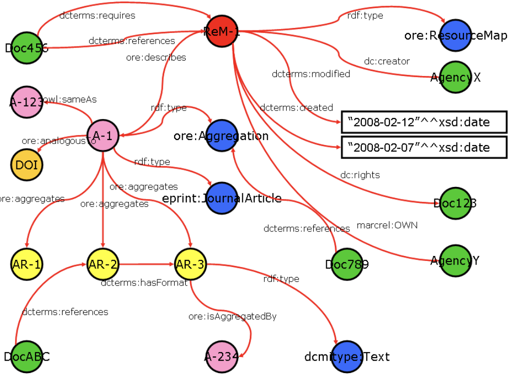
We consider now the main components that make up a VKG system, and the languages used to define them.

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)

RDF – Data is represented as a graph

The graph consists of a set of subject-predicate-object triples relating objects to other objects or values, and to classes.



Object property:

<A-1> ore:describes <ReM-1> .

Data property:

<ReM-1> :created "2008-02-07" .

Class membership:

<A-1> rdf:type :JournalArticle .

SPARQL query language

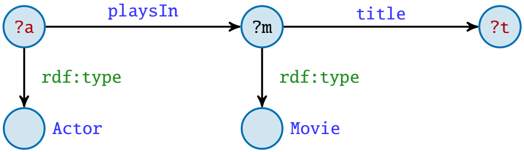
- Is the standard query language for RDF data. [W3C Rec. 2008, 2013]

```
SELECT ?a ?t
WHERE { ?a rdf:type Actor .
        ?a playsIn ?m .
        ?m rdf:type Movie .
        ?m title ?t .
      }
```

SPARQL query language

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

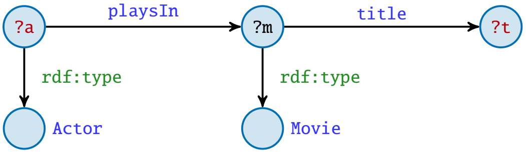
```
SELECT ?a ?t
WHERE { ?a rdf:type Actor .
        ?a playsIn ?m .
        ?m rdf:type Movie .
        ?m title ?t .
      }
```



SPARQL query language

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

```
SELECT ?a ?t
WHERE { ?a rdf:type Actor .
        ?a playsIn ?m .
        ?m rdf:type Movie .
        ?m title ?t .
      }
```



Additional language features (SPARQL 1.1):

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

SPARQL Basic Graph Patterns

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

Example: BGP

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

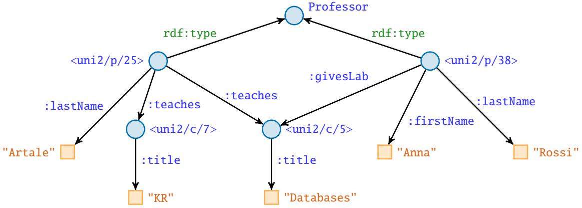
SPARQL Basic Graph Patterns

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

Example: BGP

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

When evaluated over the RDF graph



… the query returns:

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

Abbreviated syntax for Basic Graph Patterns

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

Ex.: BGP

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

Abbreviated syntax for Basic Graph Patterns

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

Ex.: BGP

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

Ex.: 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.

Ex.: BGP with projection

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

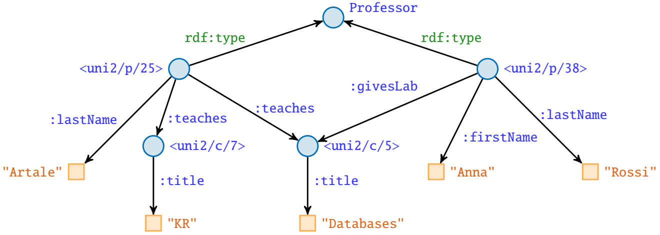
Projecting out variables in a SPARQL query

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

Ex.: BGP with projection

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

When evaluated over the RDF graph



… the query returns:

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

Anonymous variables

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

Ex.: BGP

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

Anonymous variables

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

Ex.: BGP

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

Ex.: BGP with anonymous variable

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

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

Union of Basic Graph Patterns

Example: BGPs with UNION

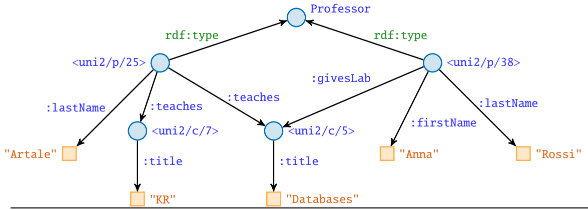
```
SELECT ?p ?ln ?c
WHERE {
  { ?p :lastName ?ln .    ?p :teaches ?c . }
  UNION
  { ?p :lastName ?ln .    ?p :givesLab ?c . }
}
```

Union of Basic Graph Patterns

Example: BGPs with UNION

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

When evaluated over



... the query returns:

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

Extending BGPs with OPTIONAL

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

Ex.: BGP with OPTIONAL

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

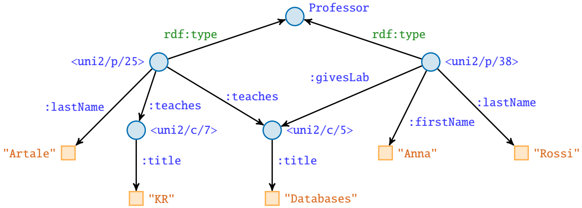
Extending BGPs with OPTIONAL

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

Ex.: BGP with OPTIONAL

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

When evaluated over the RDF graph



… the query returns:

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

`ORDER BY`, `LIMIT`, and `OFFSET`

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

ORDER BY, LIMIT, and OFFSET

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

Ex.: Ordering and limiting results

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

ORDER BY, LIMIT, and OFFSET

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

Ex.: Ordering and limiting results

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

Ex.: Multiple order comparators

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

The default is no limit, and offset 0.

`FILTER` conditions

We might want to select only those query answers respecting a condition.
This can be achieved by adding `FILTER` conditions to the query.

FILTER conditions

We might want to select only those query answers respecting a condition. This can be achieved by adding FILTER conditions to the query.

Example: BGP with a FILTER condition

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

FILTER conditions

We might want to select only those query answers respecting a condition. This can be achieved by adding FILTER conditions to the query.

Example: BGP with a FILTER condition

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

FILTER() takes an expression returning an xsd:boolean, built using:

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

SPARQL algebra

We have seen the following features of the SPARQL algebra:

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

SPARQL algebra

We have see the following features of the SPARQL algebra:

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

The overall algebra has additional features:

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

The OWL 2 QL ontology language

- OWL 2 QL is one of the three standard profiles of OWL 2.
  [W3C Rec. 2012]

- Is considered a lightweight ontology language:
  - controlled expressive power
  - efficient inference

- Optimized for accessing large amounts of data
  - Queries over the ontology can be rewritten into SQL queries over the underlying relational database (First-order rewritability).
  - Consistency of ontology and data can also be checked by executing SQL queries.

Main constructs of OWL 2 QL

Class hierarchy: `rdfs:subClassOf`

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

Main constructs of OWL 2 QL

Class hierarchy: `rdfs:subClassOf`

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

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

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

Main constructs of OWL 2 QL

Class hierarchy: `rdfs:subClassOf`

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

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

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

Domain of properties: `rdfs:domain`

Example: `:playsIn` **`rdfs:domain`** `:MovieActor .`

Main constructs of OWL 2 QL

Class hierarchy: `rdfs:subClassOf`

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

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

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

Domain of properties: `rdfs:domain`

Example: `:playsIn` **`rdfs:domain`** `:MovieActor .`

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

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

Main constructs of OWL 2 QL

Class hierarchy: `rdfs:subClassOf`

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

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

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

Domain of properties: `rdfs:domain`

Example: `:playsIn` **`rdfs:domain`** `:MovieActor .`

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

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

Range of properties: `rdfs:range`

Example: `:playsIn` **`rdfs:range`** `:Movie .`

Main constructs of OWL 2 QL

Class hierarchy: `rdfs:subClassOf`

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

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

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

Domain of properties: `rdfs:domain`

Example: `:playsIn rdfs:domain :MovieActor .`

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

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

Range of properties: `rdfs:range`

Example: `:playsIn rdfs:range :Movie .`

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

$\Longrightarrow$ `<movie/3> rdf:type :Movie .`

Other constructs of OWL 2 QL

Class disjointness: `owl:disjointWith`

Example: `:Actor owl:disjointWith :Movie .`

Other constructs of OWL 2 QL

Class disjointness: `owl:disjointWith`

Example: `:Actor` **`owl:disjointWith`** `:Movie .`

Inference:
```
<person/2> rdf:type :Actor .
<person/2> rdf:type :Movie .
```

$\Longrightarrow$ RDF graph inconsistent with the ontology

Other constructs of OWL 2 QL

Class disjointness: `owl:disjointWith`

Example: `:Actor` **`owl:disjointWith`** `:Movie .`

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

`<person/2> rdf:type :Movie .`

$\Longrightarrow$ RDF graph inconsistent with the ontology

Inverse properties: `owl:inverseOf`

Example: `:actsIn` **`owl:inverseOf`** `:hasActor .`

Other constructs of OWL 2 QL

Class disjointness: owl:disjointWith

Example: :Actor **owl:disjointWith** :Movie .

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

Inverse properties: owl:inverseOf

Example: :actsIn **owl:inverseOf** :hasActor .

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

Other constructs of OWL 2 QL

Class disjointness: owl:disjointWith

Example: :Actor **owl:disjointWith** :Movie .

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

Inverse properties: owl:inverseOf

Example: :actsIn **owl:inverseOf** :hasActor .

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

Property hierarchy

Property disjointness

Mandatory participation

Representing OWL 2 QL ontologies as UML class diagrams

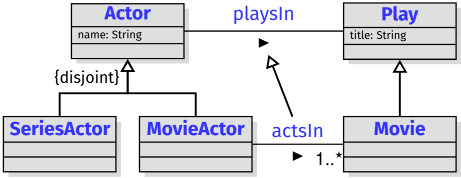
There is a close correspondence between OWL 2 QL and conceptual modeling formalisms, such as UML class diagrams and ER schemas.

```
:MovieActor rdfs:subClassOf :Actor .
:MovieActor owl:disjointWith :SeriesActor .
:actsIn rdfs:domain :MovieActor .
:actsIn rdfs:range :Movie .
:actsIn rdfs:subPropertyOf :playsIn .
...  owl:someValuesFrom ...
```

Representing OWL 2 QL ontologies as UML class diagrams

There is a close correspondence between OWL 2 QL and conceptual modeling formalisms, such as UML class diagrams and ER schemas.

`:MovieActor rdfs:subClassOf :Actor .`	subclass
`:MovieActor owl:disjointWith :SeriesActor .`	disjointness
`:actsIn rdfs:domain :MovieActor .`	domain
`:actsIn rdfs:range :Movie .`	range
`:actsIn rdfs:subPropertyOf :playsIn .`	sub-association
`... owl:someValuesFrom ...`	mandatory participation



In fact, to visualize an OWL 2 QL ontology, we can use standard UML class diagrams.

Use of mappings

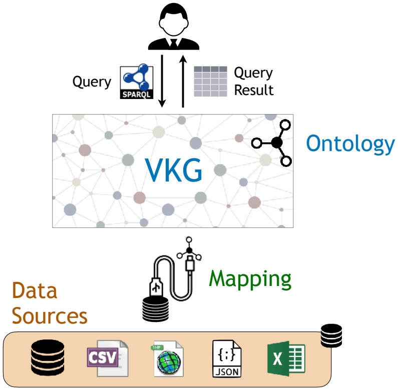
In VKGs, the mapping $\mathcal{M}$ encodes how the data $\mathcal{D}$ in the sources should be used to create the Virtual Knowledge Graph.

Use of mappings

In VKGs, the mapping $\mathcal{M}$ encodes how the data $\mathcal{D}$ in the sources should be used to create the Virtual Knowledge Graph.

VKG $\mathcal{V}$ defined from $\mathcal{M}$ and $\mathcal{D}$

- Queries are answered with respect to $\mathcal{O}$ and $\mathcal{V}$.
- The data of $\mathcal{V}$ is not materialized (it is virtual!).
- Instead, the information in $\mathcal{O}$ and $\mathcal{M}$ is used to translate queries over $\mathcal{O}$ into queries formulated over the sources.
- Advantage, compared to materialization: the graph is always up to date w.r.t. data sources.



Mapping language

The mapping consists of a set of assertions of the form

$$Q_{sql}(\vec{x}) \quad \rightsquigarrow \quad \mathbf{t}(\vec{x})\ \texttt{rdf:type}\ C$$
$$Q_{sql}(\vec{x}) \quad \rightsquigarrow \quad \mathbf{t}_1(\vec{x})\ \ p\ \ \mathbf{t}_2(\vec{x})$$

where

- $Q_{sql}(\vec{x})$ is the source query expressed in SQL,
- the right hand side is the target, consisting of a triple pattern involving a class C or a (data or object) property p, and making use of the answer variables $\vec{x}$ of the SQL query.

Mapping language

The mapping consists of a set of assertions of the form

$$Q_{sql}(\vec{x}) \quad \rightsquigarrow \quad \mathbf{t}(\vec{x}) \ \texttt{rdf:type} \ C$$
$$Q_{sql}(\vec{x}) \quad \rightsquigarrow \quad \mathbf{t}_1(\vec{x}) \ p \ \mathbf{t}_2(\vec{x})$$

where

- $Q_{sql}(\vec{x})$ is the source query expressed in SQL,
- the right hand side is the target, consisting of a triple pattern involving a class C or a (data or object) property p, and making use of the answer variables $\vec{x}$ of the SQL query.

Impedance mismatch between values in the DB and objects in the KG:
In the target, we make use of `iri`-templates $\mathbf{t}(\vec{x})$, which transform database values into IRIs (i.e., object identifiers) or literals.

Mapping language – Example

Ontology $\mathcal{O}$:

```
:actsIn rdfs:domain :MovieActor .
:actsIn rdfs:range :Movie .
:title rdfs:domain :Movie .
:title rdfs:range xsd:string .
```

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	···

Mapping language – Example

Ontology $\mathcal{O}$:

```
:actsIn rdfs:domain :MovieActor .
:actsIn rdfs:range :Movie .
:title rdfs:domain :Movie .
:title rdfs:range xsd:string .
```

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	···

Mapping language – Example

Ontology $\mathcal{O}$:

```
:actsIn rdfs:domain :MovieActor .
:actsIn rdfs:range :Movie .
:title rdfs:domain :Movie .
:title rdfs:range xsd:string .
```

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	···

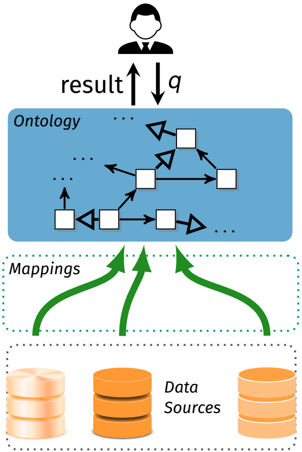
The mapping $\mathcal{M}$ applied to database $\mathcal{D}$ generates the (virtual) knowledge graph $\mathcal{V} = \mathcal{M}(\mathcal{D})$:

```
: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 .
```

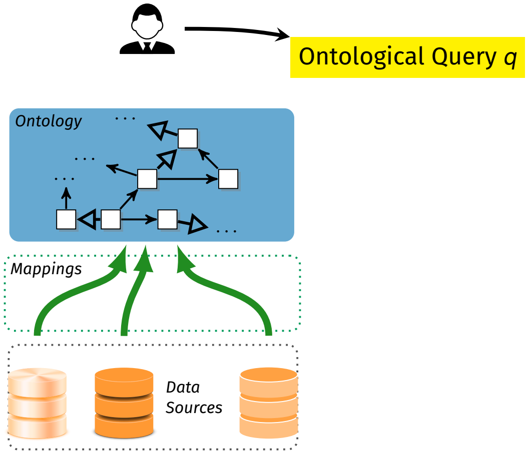
Outline

1. Challenges in Data Access
2. A Quick History of VKGs
3. Ontop
4. The VKG Framework
5. Query Answering in VKGs

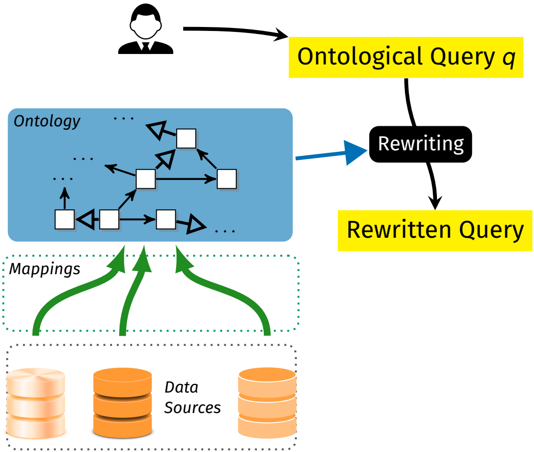
Query answering via query reformulation – Conceptual framework



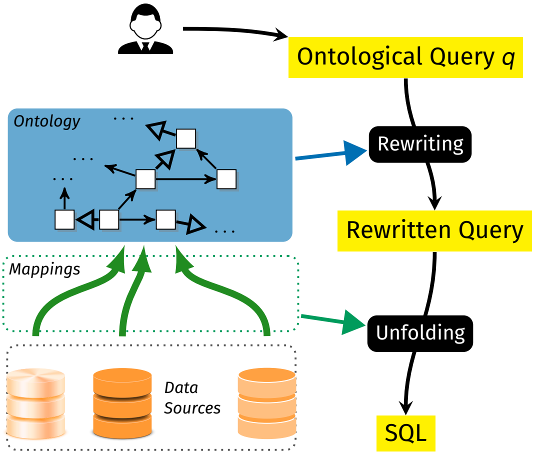
Query answering via query reformulation – Conceptual framework



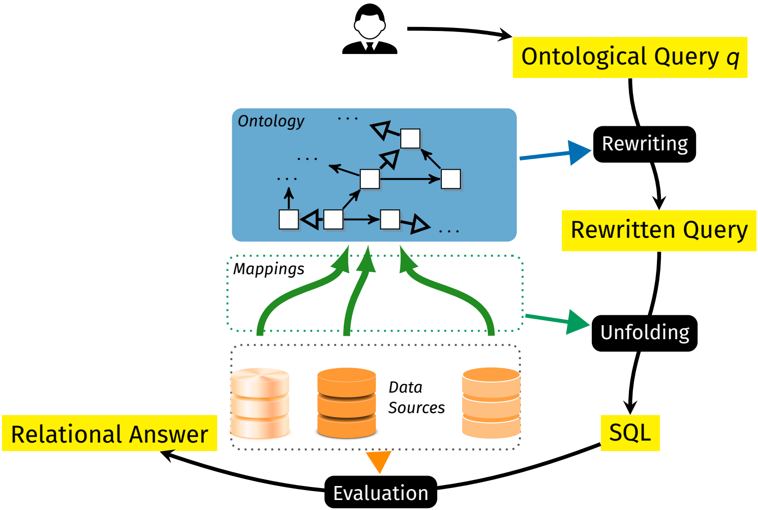
Query answering via query reformulation – Conceptual framework



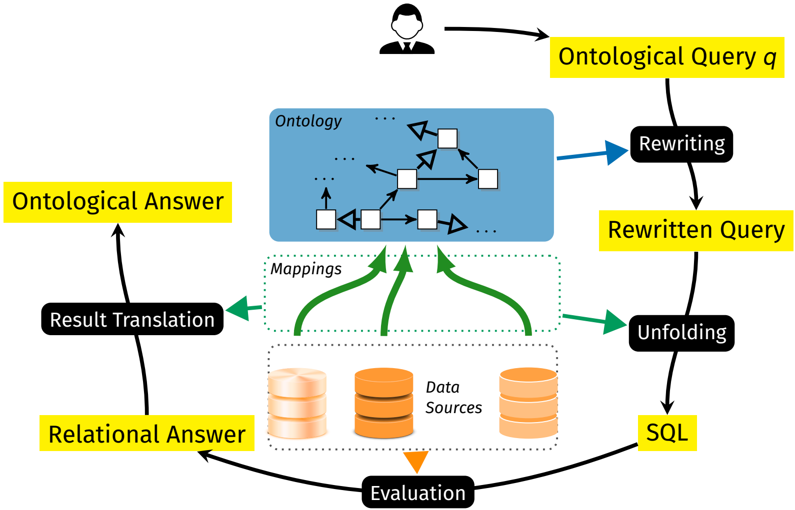
Query answering via query reformulation – Conceptual framework



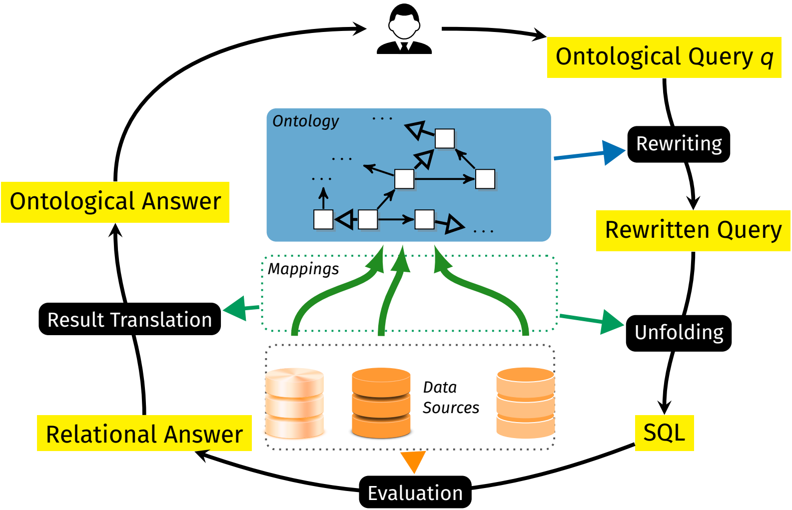
Query answering via query reformulation – Conceptual framework



Query answering via query reformulation – Conceptual framework



Query answering via query reformulation – Conceptual framework



Rewriting step

The Rewriting Step deals with the knowledge encoded in 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 class hierarchies.

Dealing with hierarchies

Suppose that every MovieActor is an Actor, i.e.,

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

and that keanu is a MovieActor: `:keanu rdf:type :MovieActor .`

Dealing with hierarchies

Suppose that every MovieActor is an Actor, i.e.,

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

and that `keanu` is a MovieActor: `:keanu rdf:type :MovieActor .`

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

```
SELECT ?x WHERE { ?x a :Actor . }
```

Dealing with hierarchies

Suppose that every MovieActor is an Actor, i.e.,

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

and that keanu is a MovieActor: `:keanu rdf:type :MovieActor .`

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

```
SELECT ?x WHERE { ?x a :Actor . }
```

The answer should be keanu, since being a MovieActor, he is also an Actor.

Dealing with hierarchies

Suppose that every MovieActor is an Actor, i.e.,

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

and that keanu is a MovieActor: `:keanu rdf:type :MovieActor .`

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

```
SELECT ?x WHERE { ?x a :Actor . }
```

The answer should be keanu, since being a MovieActor, he is also an Actor.

In fact, the query rewriting algorithm applies the above inclusion axiom as a kind of rule from right to left, and rewrites the query into a UNION query:

```
SELECT DISTINCT ?x
WHERE {
  { ?x a :Actor . }  UNION  { ?x a :MovieActor . }
}
```

Contributions of rewriting and unfolding

By computing the rewriting $\boldsymbol{q}_{\mathrm{r}}$ of $\boldsymbol{q}$ w.r.t. $\mathcal{O}$ and its unfolding $\boldsymbol{q}_{\mathrm{unf}}$ w.r.t. $\mathcal{M}$, the resulting query $\boldsymbol{q}_{\mathrm{unf}}$ might become too large and costly to execute over $\mathcal{D}$.

Contributions of rewriting and unfolding

By computing the rewriting $\boldsymbol{q}_{\mathsf{r}}$ of $\boldsymbol{q}$ w.r.t. $\mathcal{O}$ and its unfolding $\boldsymbol{q}_{\mathsf{unf}}$ w.r.t. $\mathcal{M}$, the resulting query $\boldsymbol{q}_{\mathsf{unf}}$ might become too large and costly to execute over $\mathcal{D}$.

Let's consider how rewriting and unfolding contribute to query answers:

- In principle, evaluating $\boldsymbol{q}_{\mathsf{unf}}$ over $\mathcal{D}$, gives the same result as evaluating $\boldsymbol{q}_{\mathsf{r}}$ over the RDF graph $\mathcal{V} = \mathcal{M}(\mathcal{D})$ extracted through $\mathcal{M}$ from $\mathcal{D}$.
- Instead, the rewriting impacts query answers in two ways:
  (1) through the rewriting w.r.t. class and property hierarchies, i.e., C_1 `rdfs:subClassOf` C_2, p_1 `rdfs:subPropertyOf` p_2;
  (2) through the rewriting taking into account existential reasoning, i.e., `owl:someValuesFrom` in the right-hand side of inclusion assertions.

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 $\mathcal{V}_{sat}$, 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 inclusions that cause existential reasoning
   $\rightsquigarrow$ tree-witness rewriting q_{tw} [Kikot, Kontchakov, and Zakharyaschev 2012]
2. We use instead class and property hierarchies to enrich the mapping $\mathcal{M}$.
   $\rightsquigarrow$ saturated mapping $\mathcal{M}_{sat}$ [Kontchakov, Rezk, et al. 2014; Rodriguez-Muro, Kontchakov, and Zakharyaschev 2013]
3. We unfold the tree-witness rewriting q_{tw} w.r.t. the saturated mapping $\mathcal{M}_{sat}$.

One can show that the resulting query is equivalent to the one obtained via ordinary rewriting w.r.t. $\mathcal{O}$ and unfolding w.r.t. $\mathcal{M}$.

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

Saturated mapping

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

For each mapping assertion in $\mathcal{M}$	and each TBox assertion in $\mathcal{O}$	we add a mapping assertion to $\mathcal{M}_{\text{sat}}$
$Q_{sql}(\vec{x}) \rightsquigarrow \mathbf{t}(\vec{x})$ `rdf:type` C_1	C_1 `rdfs:subClassOf` C_2	$Q_{sql}(\vec{x}) \rightsquigarrow \mathbf{t}(\vec{x})$ `rdf:type` C_2
$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})\ \ p\ \ \mathbf{t}_2(\vec{y})$	p `rdfs:domain` C_1	$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})$ `rdf:type` C_1
$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})\ \ p\ \ \mathbf{t}_2(\vec{y})$	p `rdfs:range` C_2	$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_2(\vec{x})$ `rdf:type` C_2
$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})\ \ p_1\ \ \mathbf{t}_2(\vec{y})$	p_1 `rdfs:subPropertyOf` p_2	$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})\ \ p_2\ \ \mathbf{t}_2(\vec{y})$

Saturated mapping

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

For each mapping assertion in $\mathcal{M}$	and each TBox assertion in $\mathcal{O}$	we add a mapping assertion to $\mathcal{M}_{\text{sat}}$
$Q_{sql}(\vec{x}) \rightsquigarrow \mathbf{t}(\vec{x})\ \texttt{rdf:type}\ C_1$	$C_1\ \texttt{rdfs:subClassOf}\ C_2$	$Q_{sql}(\vec{x}) \rightsquigarrow \mathbf{t}(\vec{x})\ \texttt{rdf:type}\ C_2$
$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})\ \ p\ \ \mathbf{t}_2(\vec{y})$	$p\ \texttt{rdfs:domain}\ C_1$	$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})\ \texttt{rdf:type}\ C_1$
$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})\ \ p\ \ \mathbf{t}_2(\vec{y})$	$p\ \texttt{rdfs:range}\ C_2$	$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_2(\vec{x})\ \texttt{rdf:type}\ C_2$
$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})\ \ p_1\ \ \mathbf{t}_2(\vec{y})$	$p_1\ \texttt{rdfs:subPropertyOf}\ p_2$	$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})\ \ p_2\ \ \mathbf{t}_2(\vec{y})$

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

Saturated mapping

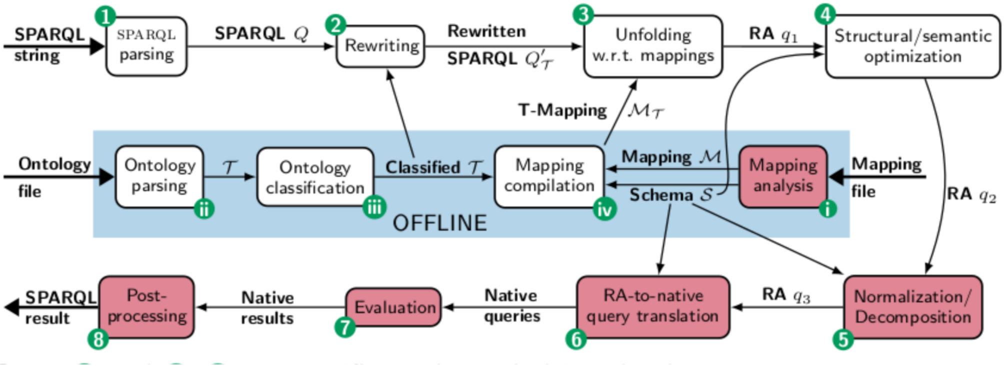
Intuitively, the saturated mapping $\mathcal{M}_{\text{sat}}$ is the composition of $\mathcal{M}$ and $\mathcal{O}$.

For each mapping assertion in $\mathcal{M}$	and each TBox assertion in $\mathcal{O}$	we add a mapping assertion to $\mathcal{M}_{\text{sat}}$
$Q_{sql}(\vec{x}) \rightsquigarrow \mathbf{t}(\vec{x})$ `rdf:type` C_1	C_1 `rdfs:subClassOf` C_2	$Q_{sql}(\vec{x}) \rightsquigarrow \mathbf{t}(\vec{x})$ `rdf:type` C_2
$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})\ \ p\ \ \mathbf{t}_2(\vec{y})$	p `rdfs:domain` C_1	$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})$ `rdf:type` C_1
$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})\ \ p\ \ \mathbf{t}_2(\vec{y})$	p `rdfs:range` C_2	$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_2(\vec{x})$ `rdf:type` C_2
$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})\ \ p_1\ \ \mathbf{t}_2(\vec{y})$	p_1 `rdfs:subPropertyOf` p_2	$Q_{sql}(\vec{x},\vec{y}) \rightsquigarrow \mathbf{t}_1(\vec{x})\ \ p_2\ \ \mathbf{t}_2(\vec{y})$

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.

Implementation of query answering in *Ontop*



ONTOPIC

We now switch to the practical part
with Ontopic Studio,
followed by hands-on sessions.

Part II

appendix

Outline of Part 2

References I

[1] Diego Calvanese, Pietro Liuzzo, Alessandro Mosca, Jose Remesal, Martin Rezk, and Guillem Rull. "Ontology-Based Data Integration in EPNet: Production and Distribution of Food During the Roman Empire". In: *Engineering Applications of Artificial Intelligence* 51 (2016), pp. 212–229. DOI: `10.1016/j.engappai.2016.01.005`.

[2] Stanislav Kikot, Roman Kontchakov, and 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.

References II

[3] Roman Kontchakov, Martin Rezk, Mariano Rodriguez-Muro, Guohui Xiao, and 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.

[4] Roman Kontchakov and 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.

References III

[5] Mariano Rodriguez-Muro, Roman Kontchakov, and 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.

[6] Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, and 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.

References IV

[7] Guohui Xiao, Linfang Ding, Benjamin Cogrel, and Diego Calvanese. "Virtual Knowledge Graphs: An Overview of Systems and Use Cases". In: *Data Intelligence* 1.3 (2019), pp. 201–223. DOI: 10.1162/dint_a_00011.