Integrating Data through Virtual Knowledge Graphs with Ontop

Diego Calvanese, Benjamin Cogrel, Guohui Xiao
Ontopic s.r.l.

Knowledge Graph Conference, 3 May 2021

Data integration

Databases are great!

They let us manage efficiently huge amounts of data ...

... assuming you have put them all into your schema

Data integration

Databases are great!
They let us manage efficiently huge amounts of data ...
... assuming you have put them all into your schema

However, the reality is much more involved and heterogeneous:

- Data sets were created independently
- Data are often stored across different sources
- Data sources are controlled by different people / organizations

Data integration

Databases are great!
They let us manage efficiently huge amounts of data …
… assuming you have put them all into your schema

However, the reality is much more involved and heterogeneous:

- Data sets were created independently
- Data are often stored across different sources
- Data sources are controlled by different people / organizations

The goal of data integration is to put together different data sources, created for different purposes, and controlled by different people, making them accessible in a uniform way.

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

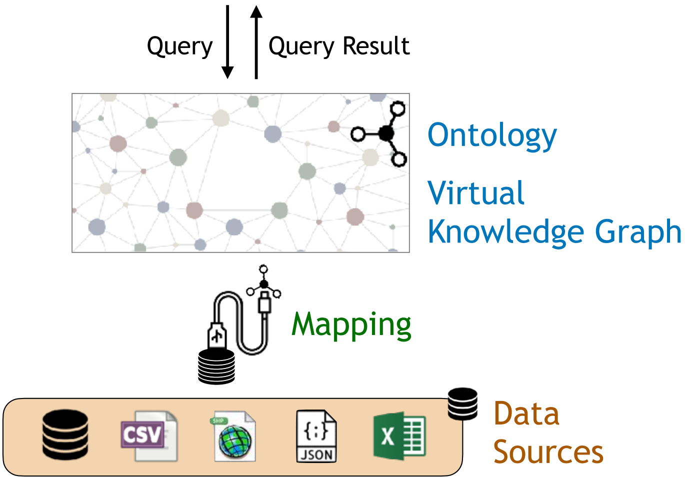
Movie (title, director, year, genre)
Actor (title, name)
Plays (movie, location, startTime)
Review (title, rating, description)

How to address heterogeneity?

We use a combination of three key ideas:

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

Virtual Knowledge Graph (VKG) architecture



Why a mapping?

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 can be used for query optimization

Why a KG for the global schema?

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

A KG, 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 / data sources

Why virtualization?

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

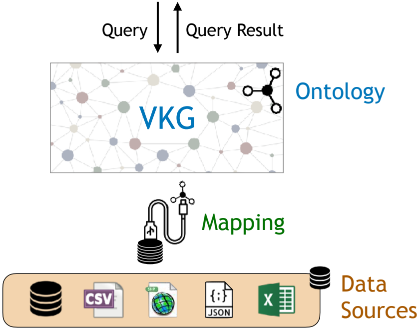
In the purely virtual approach, instead:

- The data stays in the sources and is only accessed at query time
- There is no need to construct a large and potentially costly materialization and to 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
- Better supports an incremental approach to integration

Components of the VKG architecture

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

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

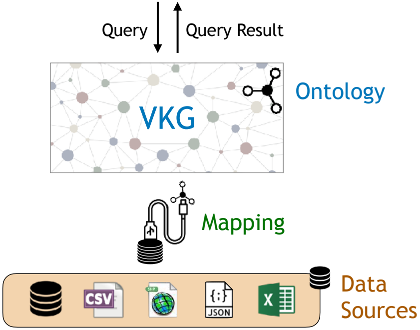


ONTOPIC

Components of the VKG architecture

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

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



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. Data Integration
2. A Quick History of VKGs
3. Ontop
4. Use Cases
5. The VKG Framework
6. Input Dataset Handling
7. Hands-on

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 $\leadsto$ mostly theoretical
- **2005** Families of lightweight ontology languages (or Description Logics) $\leadsto$ 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. Data Integration
2. A Quick History of VKGs
3. Ontop
4. Use Cases
5. The VKG Framework
6. Input Dataset Handling
7. Hands-on

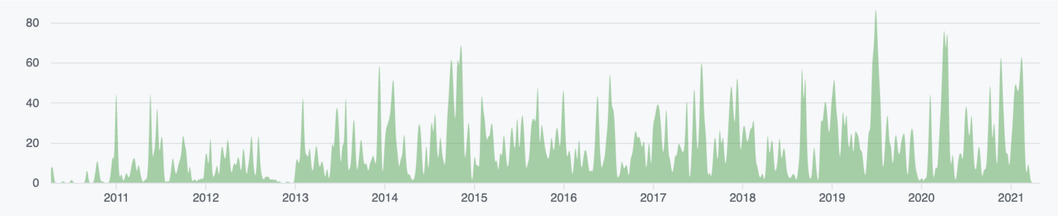
The *Ontop* system

ontop

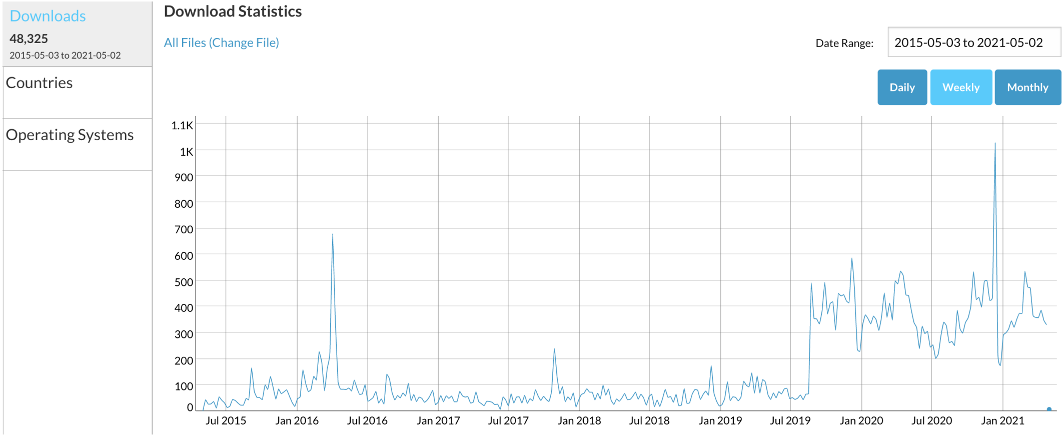
https://ontop-vkg.org/

- 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



Ontop downloads



Outline

1. Data Integration
2. A Quick History of VKGs
3. Ontop
4. Use Cases
5. The VKG Framework
6. Input Dataset Handling
7. Hands-on

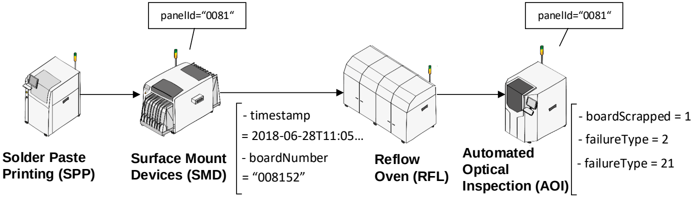
Use cases of Ontop

- Adopted in many academic and industrial use cases.[1]
- Some application areas:
  - 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

[1] Guohui Xiao, Linfang Ding, Benjamin Cogrel, and Diego Calvanese. Virtual knowledge graphs: An overview of systems and use cases. *Data Intelligence*, 1:201–223, 2019.

ONTOPIC

Failure detection for Surface Mounting Process pipeline in Bosch[2]



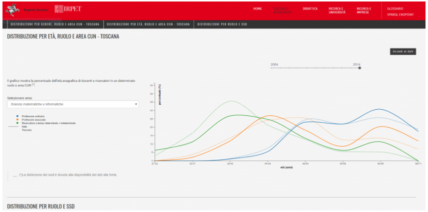
- Failure detection fundamentally relies on the integration and analysis of data generated in different phases
- Such machines come from different suppliers and rely on distinct formats

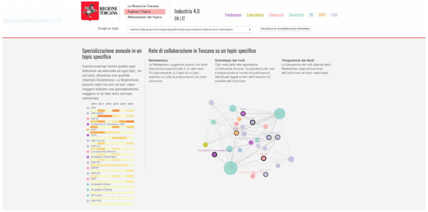
[2] E Güzel Kalaycı, I Grangel Gonalez, F Lösch, G Xiao, A ul Mehdi, E Kharlamov, and D Calvanese. Semantic integration of Bosch manufacturing data using virtual knowledge graphs. In *ISWC*, 2020.

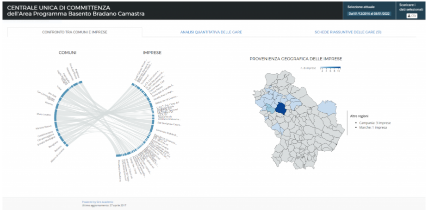
Use cases of Ontop

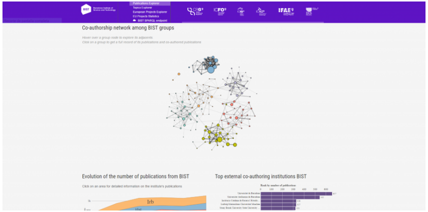
- Adopted in many academic and industrial use cases.
- Some application areas:
  - Industry 4.0
    - Many vendors / historical data of exploration campaigns
    - Examples: Equinor, Siemens, Bosch
  - Analytical / BI
    - Combine internal data, manual processes (e.g., Excel) and external data
    - 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

Toscana Open Research









http://www.toscanaopenresearch.it/en/

A large European university

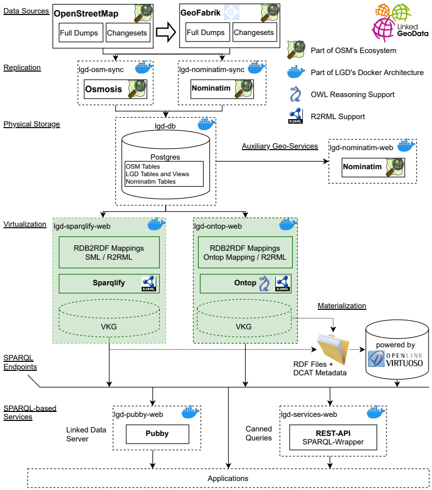
- Internal data
  - Research funding, HR, teaching, etc.
  - Redundant applications due to the merge of several universities
  - Operational data store and data warehouse
  - Many processes are still using Excel
- External data
  - Open Data (from the ministry, EU commission and public initiatives)
  - Commercial bibliometric data
  - Mainly for benchmarking

Use cases of Ontop

- Adopted in many academic and industrial use cases.
- Some application areas:
  - 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

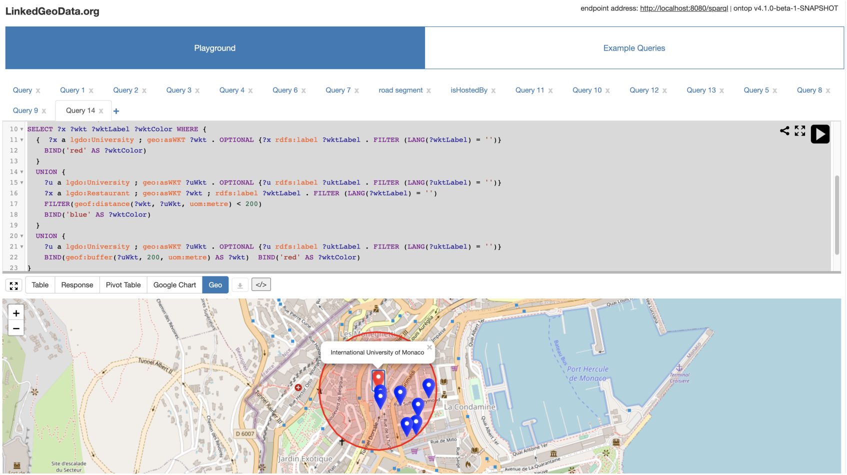
LinkedGeoData.org

- LGD converts OpenStreetMap to RDF
- one of the most important Geospatial Knowledge Graphs
- The next version of LGD will be based on Ontop
- ... in collaboration with University of Leipzig



ONTOPIC

LinkedGeoData.org



ONTOPIC

VKG over the South Tyrolean Open Data Hub (ODH)

`https://sparql.opendatahub.bz.it/`

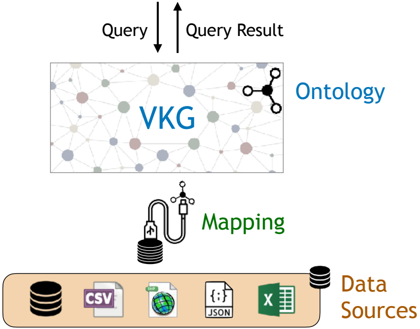
- ODH publishes tourism, mobility, and weather data from different providers through a JSON-based Web API
- The backend relies on PostgreSQL databases
- Joint project between Ontopic and NOI Techpark on extending ODH with a VKG

The demos and hands-on of this tutorial are adapted from this use case.

Outline

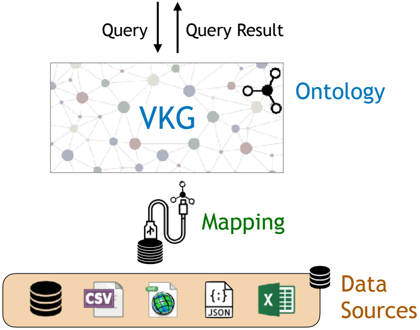
Components of the VKG architecture

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



ONTOPIC

Components of the VKG architecture



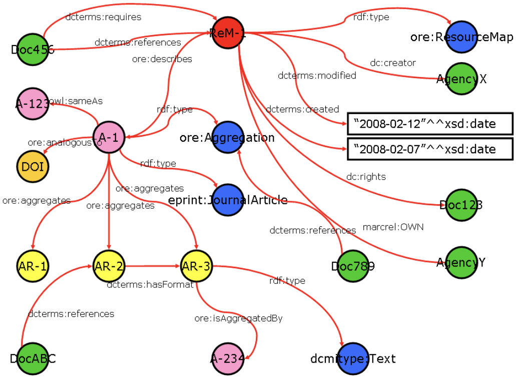
We consider now the main components that make up a VKG system, and the languages used to define them.

The W3C has standardized languages that are suitable for VKGs:

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

RDF – Data is represented as a graph

The graph consists of a set of subject-predicate-object triples:



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

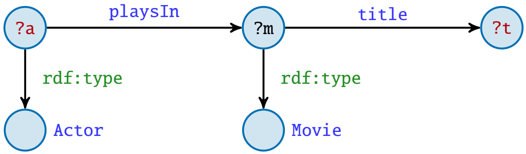
- Is the standard query language for RDF data. [W3C Rec. 2008, 2013]

```
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.

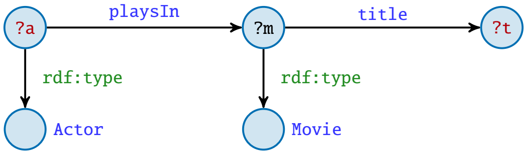
```
SELECT ?a ?t
WHERE { ?a rdf:type Actor .
        ?a playsIn ?m .
        ?m rdf:type Movie .
        ?m title ?t .
      }
```



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)

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 .`

ONTOPIC

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

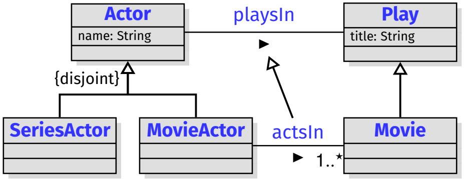
There is a close correspondence between OWL 2 QL and conceptual modeling formalisms, such as UML class diagrams and ER schemas.

```
: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

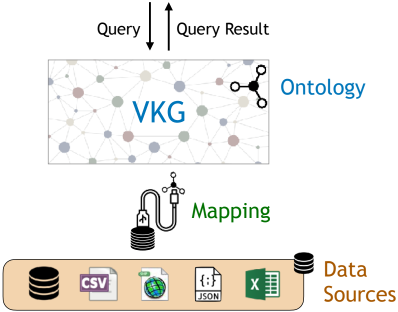
In VKGs, the mapping $\mathcal{M}$ encodes how the data $\mathcal{D}$ in the sources should be used to create the virtual knowledge graph.

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.

Virtual knowledge graph $\mathcal{V}$ 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

$$Q_{sql}(\vec{x}) \quad \rightsquigarrow \quad \mathbf{t}(\vec{x}) \ \mathtt{rdf{:}type} \ \ C$$

$$Q_{sql}(\vec{x}) \quad \rightsquigarrow \quad \mathbf{t}_1(\vec{x}) \ \ p \ \ \mathbf{t}_2(\vec{x})$$

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 $\mathcal{D}$:

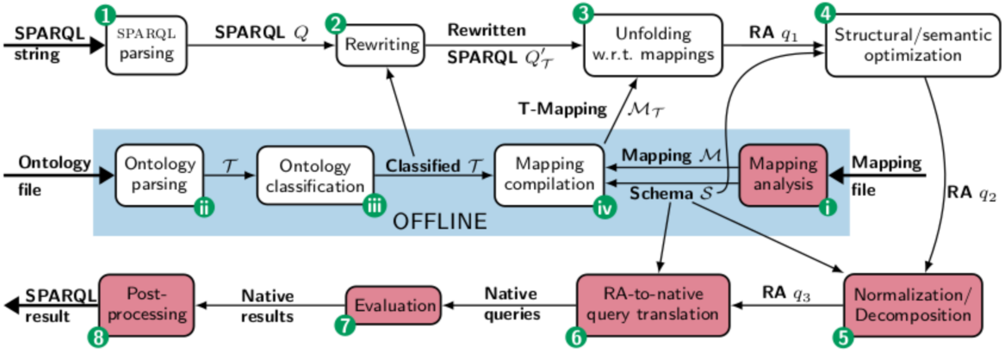
MOVIE				
mcode	*mtitle*	*myear*	*type*	···
5118	The Matrix	1999	m	···
8234	Altered Carbon	2018	s	···
2281	Blade Runner	1982	m	···

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 .
```

Virtual approach for query answering in *Ontop*



Rewriting step

The rewriting Step 2 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 .`

ONTOPIC

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 . }
}
```


ONTOPIC

Demo: Basic usage of Ontop

```
$ git clone \
    https://github.com/ontopic-vkg/destination-tutorial \
    --config core.autocrlf=input # important for Windows
$ cd destination-tutorial
$ docker-compose -f docker-compose.solution.yml up
```

1. Check the database in DBeaver
2. Open `vkg/dest-solution.ttl` in Protégé
3. Open `http://localhost:8080` in the browser

Outline

1. Data Integration
2. A Quick History of VKGs
3. Ontop
4. Use Cases
5. The VKG Framework
6. Input Dataset Handling
7. Hands-on

Direct input for Ontop ("sources")

- Transactional database in production (not so often)
- Physical replica
- Logical replica: allows for basic transformations
  - Flattening JSON structure
  - Adding geospatial indexes
  - Merging different databases (e.g. managed by different teams)
- Operational data store
- Data warehouse

Mediated input for Ontop

- Data lake: files (e.g., CSV, JSON)
  - Through Denodo or Dremio
  - Populated by data pipelines
  - Provided by non-IT people (first iterations)
- WebAPI
  - Through Denodo
  - Often comes with querying pattern restrictions
- More than one source for the same Ontop instance
  - Through Denodo, Dremio, or Teiid (coming soon)

Data federation with Dremio

- Supports data lakes, relational databases and several NoSQL systems
- Open source (Apache 2.0)
- Distributed query processing by pushing sub-queries to the sources
- Acceleration though “reflections” when needed
  - Particularly powerful for aggregation queries (e.g., slicing/dicing)
  - Often considered as a second step, for accelerating some queries
  - Make sure to check first that no integrity constraint is missing!
  - Materialization remains at the relational level
- Limited set of functions
  - E.g., does not support geospatial functions
- Does not expose integrity constraints
  - They have to be specified externally

Data federation with Dremio

Demo

Pros of the virtual approach to KGs

- Not being required to move data allows for fast iterations
- Reuses the existing infrastructure, methods and expertise present in the company
- Often perceived less intrusive to admins than a new database technology they don't know
- Most inner and left joins can be eliminated at the SQL level
- Materialization concerns come later, e.g., for accelerating some queries
- Reasoning costs are usually very low

Cons of the virtual approach to KGs

- Requires paying more attention to mapping quality and integrity constraints
- Non-RDF materialization, when needed, is at the moment still fairly manual
- Meta-queries can be challenging (new optimizations to come)
- Less expressive reasoning capabilities (in the absence of advanced post-processing capabilities)
- Dealing with RDF dumps implies at the moment SPARQL federation
- No native support for graph analytics (has to be done externally)

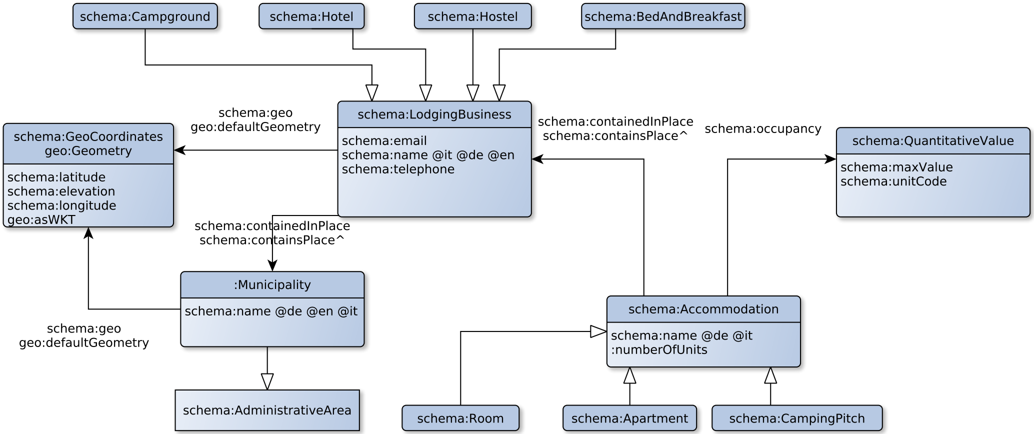
Outline

Destination tutorial

`https://github.com/ontopic-vkg/destination-tutorial`

- Focused on the mapping design
- Ontology already provided
- SPARQL endpoint and database handled by Docker-compose
- Guidance for specifying the mapping will be published in the coming days

Lodging businesses and municipalities



Weather stations