Ontology-based Data Access
A Tutorial on Query Reformulation and Optimization

Diego Calvanese

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

Seminar on Ontology Research in Brazil (ONTOBRAS)
São Paulo, Brazil, 1–3 October 2018
Outline

1. Query rewriting wrt an OWL 2 QL ontology
2. Mapping specification
3. Saturation and optimization of the mapping
4. Query reformulation and optimization
Outline

1. Query rewriting wrt an OWL 2 QL ontology
2. Mapping specification
3. Saturation and optimization of the mapping
4. Query reformulation and optimization
To compute the certain answers to a SPARQL query \( q \) over an OBDA instance \( O = \langle P, D \rangle \), with \( P = \langle T, S, M \rangle \):

1. Compute the perfect rewriting of \( q \) w.r.t. \( T \).
2. Unfold the perfect rewriting wrt the mapping \( M \).
3. Optimize the unfolded query, using database constraints.
4. Evaluate the resulting SQL query over \( D \).

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

The rewriting Step 1 deals with the objects that are existentially implied by the axioms of the ontology.
Example of existential reasoning

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

\[ \text{GraduateStudent} \sqsubseteq \exists \text{isSupervisedBy}.\text{Professor} \]

and john is a graduate student: \[ \text{GraduateStudent}(\text{john}) \].

What is the answer to the following query?

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

The answer should be \text{john}, even though we don’t know who is John’s supervisor (under existential reasoning).
Existential reasoning and query rewriting

Canonical model

Every consistent $DL$-Lite KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ has a **canonical model** $I_\mathcal{K}$, which gives the right answers to all CQs, i.e., $\text{cert}(q, \mathcal{K}) = \text{ans}(q, I_\mathcal{K})$.

- The core part can be handled by **saturating the mapping**.
- The anonymous part can be handled by **Tree-witness rewriting**.
Example of existential reasoning (continued)

Using the (tree witness) rewriting algorithm, the query

\[ q(x) \leftarrow is\text{SupervisedBy}(x, y), Professor(y) \]

is rewritten to a union of two conjunctive queries (or a SPARQL union query):

\[ q(x) \leftarrow is\text{SupervisedBy}(x, y), Professor(y) \]
\[ q(x) \leftarrow GraduateStudent(x) \]

Therefore, over the Abox \textit{GraduateStudent(john)}, the rewritten query returns \textit{john} as an answer.

\textbf{Note}: In \textit{Ontop}, if one wants to answer queries by performing existential reasoning, the tree-witness rewriting algorithm needs to be switched on explicitly.
The \textit{PerfectRef} algorithm for query rewriting

To illustrate Step 1 of the query reformulation algorithm, we briefly describe \textit{PerfectRef}, a simple query rewriting algorithm that requires to iterate over:

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

The perfect rewriting of $q$ is still a SPARQL query involving \texttt{UNION}.

\textit{Note}: disjointness assertions play a role in ontology satisfiability, but can be ignored during query rewriting (i.e., we have \texttt{separability}).
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 \( \text{FullProf} \sqsubseteq \text{Prof} \) corresponds to the logic programming rule

\[
\text{Prof}(z) \leftarrow \text{FullProf}(z).
\]

Consider the query

\[
q(x) \leftarrow \text{Prof}(x).
\]

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

\[
q(x) \leftarrow \text{FullProf}(x).
\]

This query is added to the input query, and contributes to the perfect rewriting.
Query rewriting (cont’d)

Example
Consider the query

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

and the inclusion assertion

$$\exists \text{teaches}^{-} \sqsubseteq \text{Course}$$

as a logic programming rule:

$$\text{Course}(z_2) \leftarrow \text{teaches}(z_1, z_2).$$

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

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

Example
Consider now the query

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

and the inclusion assertion

$$\text{FullProf} \sqsubseteq \exists \text{teaches}$$

as a logic programming rule:

$$\text{teaches}(z, f(z)) \leftarrow \text{FullProf}(z).$$

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

$$q(x) \leftarrow \text{FullProf}(x).$$
Conversely, for the query \( q(x) \leftarrow teaches(x, \text{databases}) \) and the same inclusion assertion as before as a logic programming rule:

\[
\text{FullProf} \sqsubseteq \exists \text{teaches} \quad teaches(z, f(z)) \leftarrow \text{FullProf}(z)
\]

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

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

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:

\[
q(x, y) \leftarrow teaches(x, y).
\]
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

\[ q(x) \leftarrow teaches(x, y), \text{Course}(y) \]

and the inclusion assertion

\[ FullProf \sqsubseteq \exists \text{teaches} \]

as a logic programming rule:

\[ teaches(z, f(z)) \leftarrow FullProf(z). \]

The **inclusion assertion** above does **not** apply to the atom \( teaches(x, y) \).
Example

Consider now the query

\( q(x) \leftarrow teaches(x, y), teaches(z, y) \)

and the inclusion assertion

\( FullProf \sqsubseteq \exists teaches \)

as a logic rule:

\( teaches(z, f(z)) \leftarrow FullProf(z) \).

This inclusion assertion does not apply to \( teaches(x, y) \) or \( teaches(z, y) \), 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 \( teaches(x, y) \) and \( teaches(z, y) \). This rewriting step is called **reduce**, and produces the query

\( q(x) \leftarrow teaches(x, y) \).

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

\( q(x) \leftarrow FullProf(x) \). \)
Query rewriting – Summary

To compute the perfect rewriting of a query $q$, 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 $\mathcal{T}$ as follows:

  - $A_1 \sqsubseteq A_2 \quad \Rightarrow \quad \ldots, A_2(x), \ldots \Rightarrow \ldots, A_1(x), \ldots$
  - $\exists P \sqsubseteq A \quad \Rightarrow \quad \ldots, A(x), \ldots \Rightarrow \ldots, P(x, \_), \ldots$
  - $\exists P^- \sqsubseteq A \quad \Rightarrow \quad \ldots, A(x), \ldots \Rightarrow \ldots, P(\_, x), \ldots$
  - $\exists P \sqsubseteq \exists P \quad \Rightarrow \quad \ldots, P_2(x, \_), \ldots \Rightarrow \ldots, P_1(x, \_), \ldots$
  - $P_1 \sqsubseteq P_2 \quad \Rightarrow \quad \ldots, P_2(x, y), \ldots \Rightarrow \ldots, P_1(x, y), \ldots$
  - $P_1 \sqsubseteq P_2^- \quad \Rightarrow \quad \ldots, P_2(x, y), \ldots \Rightarrow \ldots, P_1(y, x), \ldots$

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

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

Each time, the result of the above step is added to the queries to be processed.

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

The UCQ resulting from this process is the **perfect rewriting** $r_{q,\mathcal{T}}$. 

---

Diego Calvanese (unibz)

Tutorial on OBDA

ONTOBRAS – 1-3/10/2018 (12/38)
Query rewriting algorithm

Algorithm \textit{PerfectRef}(Q, T_P)

\textbf{Input:} union of conjunctive queries \( Q \), set \( T_P \) of \textit{DL-Lite} inclusion assertions

\textbf{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 \( T_P \) do

if \( I \) is applicable to \( g \) then \( PR := PR \cup \{ \text{ApplyPl}(q, g, I) \} \);

for each \( g_1, g_2 \) in \( q \) do

if \( g_1 \) and \( g_2 \) unify then \( PR := PR \cup \{ \tau(\text{Reduce}(q, g_1, g_2)) \} \);

until \( PR' = PR \);

return \( PR \)

\textbf{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

**TBox:**

\[
\begin{align*}
&\text{FullProf} \sqsubseteq \text{Prof} \\
&\text{Prof} \sqsubseteq \exists \text{teaches} \\
&\exists \text{teaches} \sqsubseteq \text{Course}
\end{align*}
\]

**Corresponding rules:**

\[
\begin{align*}
&\text{Prof}(x) \leftarrow \text{FullProf}(x) \\
&\exists y(\text{teaches}(x, y)) \leftarrow \text{Prof}(x) \\
&\text{Course}(x) \leftarrow \text{teaches}(y, x)
\end{align*}
\]

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

**Perfect rewriting:**

\[
\begin{align*}
&q(x) \leftarrow \text{teaches}(x, y), \text{Course}(y) \\
&q(x) \leftarrow \text{teaches}(x, y), \text{teaches}(\_ , y) \\
&q(x) \leftarrow \text{teaches}(x, \_ ) \\
&q(x) \leftarrow \text{Prof}(x) \\
&q(x) \leftarrow \text{FullProf}(x)
\end{align*}
\]

**ABox:**

\[
\begin{align*}
&\text{teaches}(\text{jim}, \text{databases}) \\
&\text{teaches}(\text{julia}, \text{security}) \\
&\text{FullProf}(\text{jim}) \\
&\text{FullProf}(\text{nicole})
\end{align*}
\]

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

Diego Calvanese (unibz)
Query answering in *DL-Lite* – An interesting example

**TBox:**  
\[ \text{Person} \sqsubseteq \exists \text{hasFather} \]  
\[ \exists \text{hasFather}^\perp \sqsubseteq \text{Person} \]

**ABox:**  
\[ \text{Person}(\text{john}) \]

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

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

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

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

\[ \Downarrow \text{Apply } \exists \text{hasFather}^\perp \sqsubseteq \text{Person} \text{ to the atom } \text{Person}(y_2) \]

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

\[ \Downarrow \text{Unify } \text{atoms } \text{hasFather}(y_1, y_2) \text{ and } \text{hasFather}(_, y_2) \]

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

\[ \quad \Downarrow \quad \ldots \]

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

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

\[ q(x) \leftarrow \text{Person}(x) \]
Complexity of query answering in *DL-Lite*

**Query answering** for UCQs / SPARQL queries is:

- Efficiently tractable in the size of the TBox, i.e., PTIME.
- Very efficiently tractable in the size of the ABox, i.e., AC⁰.
- Exponential in the size of the query, more precisely NP-complete.

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

---

Can we go beyond *DL-Lite*?

Essentially no! By adding essentially any additional DL constructor we lose first-order rewritability and hence these nice computational properties.
Outline

1. Query rewriting wrt an OWL 2 QL ontology
2. Mapping specification
3. Saturation and optimization of the mapping
4. Query reformulation and optimization
Impedance mismatch

We need to address the **impedance mismatch** problem

- In *relational databases*, information is represented as tuples of *values*.
- In *ontologies*, information is represented using both *objects* and values . . .
  - . . . with objects playing the main role, . . .
  - . . . and values playing a subsidiary role as fillers of object attributes.

Proposed solution:

- We specify how to construct from the data values in the relational sources the (abstract) objects that populate the data layer of the ontology.
- This specification is embedded in the mappings between the data sources and the ontology.

*Note*: the **data layer** (typically) is only *virtual*, since the objects are not materialized at the level of the ontology.
Solution to the impedance mismatch problem

We need to define a **mapping language** that allows for specifying how to transform data values into abstract objects:

- Each mapping assertion maps:
  - a query that retrieves values from a data source to . . .
  - a set of atoms specified over the ontology.

- Basic idea: use **Skolem functions** (or more concretely, **pattern templates**) in the atoms over the ontology to “generate” the objects from the data values.

- Semantics of mappings:
  - Objects are denoted by terms (of exactly one level of nesting).
  - Different terms denote different objects (i.e., we make the unique name assumption on terms).
Impedance mismatch – Example

Actual data is stored in a DB:
- An employee is identified by her SSN.
- A project is identified by its name.

\[ D_1[SSN: String, PrName: String] \]
Employees and projects they work for

\[ D_2[Code: String, Salary: Int] \]
Employee’s code with salary

\[ D_3[Code: String, SSN: String] \]
Employee’s Code with SSN

Intuitively:
- An employee should be created from her SSN: \( \text{pers}(SSN) \)
- A project should be created from its name: \( \text{proj}(PrName) \)
Creating object identifiers

We need to associate to the data in the tables objects in the ontology.

- We introduce an alphabet $\Lambda$ of **function symbols**, each with an associated arity.
- To denote values, we use value constants from an alphabet $\Gamma_v$.
- To denote objects, we use **object terms** instead of object constants.
  - An object term has the form $f(d_1, \ldots, d_n)$, with $f \in \Lambda$, and each $d_i$ a value constant in $\Gamma_v$.
  - Concretely, the object terms are obtained by instantiating the patterns with values from the database.

**Example**

- If a person is identified by her **SSN**, we can introduce a function symbol $\text{pers}/1$. If $\text{VRD56B25}$ is a SSN, then $\text{pers}(\text{VRD56B25})$ denotes a person.
- If a person is identified by her **name** and **dateOfBirth**, we can introduce a function symbol $\text{pers}/2$. Then $\text{pers}(\text{Vardi}, 25/2/56)$ denotes a person.
Mapping assertions

Mapping assertions are used to extract the data from the DB to populate the ontology.

We make use of **variable terms**, which are like object terms, but with variables instead of values as arguments of the functions.

A **mapping assertion** between a database with schema $S$ and an ontology $O$ has the form

$$\Phi(\vec{x}) \leadsto \Psi(\vec{t}, \vec{y})$$

where

- $\Phi$ is an arbitrary SQL query of arity $n > 0$ over $S$;
- $\Psi$ is a conjunctive query over $O$ of arity $n' > 0$ without existentially quantified variables;
- $\vec{x}, \vec{y}$ are variables, with $\vec{y} \subseteq \vec{x}$;
- $\vec{t}$ are variable terms of the form $f(\vec{z})$, with $f \in \Lambda$ and $\vec{z} \subseteq \vec{x}$.
Mapping assertions – Example

**Employee**

<table>
<thead>
<tr>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>empCode</td>
</tr>
<tr>
<td>salary</td>
</tr>
</tbody>
</table>

**Project**

<table>
<thead>
<tr>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>projectName</td>
</tr>
</tbody>
</table>

1..* worksFor 1..*

\[
\begin{align*}
D_1[SSN: String, PrName: String] & \\
& \text{Employees and Projects they work for} \\
D_2[Code: String, Salary: Int] & \\
& \text{Employee’s code with salary} \\
D_3[Code: String, SSN: String] & \\
& \text{Employee’s code with SSN}
\end{align*}
\]

\[m_1: \text{SELECT SSN, PrName FROM } D_1\]

\[\mapsto \text{Employee(pers(SSN)), Project(proj(PrName)), projectName(proj(PrName), PrName), worksFor(pers(SSN), proj(PrName))}\]

\[m_2: \text{SELECT SSN, Salary FROM } D_2, D_3 \]

\[\text{WHERE } D_2.\text{Code} = D_3.\text{Code}\]

\[\mapsto \text{Employee(pers(SSN)), salary(pers(SSN), Salary)}\]
Concrete 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/](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](http://d2rq.org/d2rq-language)  
\(^2\)[http://sparqlify.org/wiki/Sparqlification_mapping_language](http://sparqlify.org/wiki/Sparqlification_mapping_language)  
\(^3\)[https://github.com/ontop/ontop/wiki/ontopOBDAModel#Mapping_axioms](https://github.com/ontop/ontop/wiki/ontopOBDAModel#Mapping_axioms)
To formalize OBDA, we distinguish between the intensional and the extensional level information.

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

- $O$ is the (intensional level of an) ontology. We consider ontologies formalized in description logics (DLs), hence the intensional level is a DL TBox.
- $S$ is a (possibly federated) relational database schema for the data source(s), possibly with constraints;
- $M$ is a set of mapping assertions between $O$ and $S$.

**An OBDA instance** is a pair $\mathcal{I} = \langle \mathcal{P}, D \rangle$, where

- $\mathcal{P} = \langle O, M, S \rangle$ is an OBDA specification, and
- $D$ is a relational database compliant with $S$. 
In an OBDA instance $\mathcal{J} = \langle\langle O, M, S \rangle, D \rangle$, the mapping $M$ encodes how the data $D$ in the source(s) $S$ should be used to populate the elements of $O$.

**Virtual data layer**

The data $D$ and the mapping $M$ define a virtual data layer $V = M(D)$

- Queries are answered w.r.t. $O$ and $V$.
- We do not really materialize the data of $V$ (it is virtual!).
- Instead, the intensional information in $O$ and $M$ is used to translate queries over $O$ into queries formulated over $S$. 
Virtual data layer – Example

\[
\begin{align*}
D_1 & \quad D_2 & \quad D_3 \\
\text{SSN} & \quad \text{PrName} & \quad \text{Code} & \quad \text{Salary} & \quad \text{Code} & \quad \text{SSN} \\
23AB & \quad \text{optique} & \quad e23 & \quad 1500 & \quad e23 & \quad 23AB \\
\end{align*}
\]

\[m_1: \quad \text{SELECT SSN, PrName} \quad \mapsto \quad \text{Employee}(\text{pers}(\text{SSN})), \quad \text{Project}(\text{proj}(\text{PrName})), \quad \text{projectName}(\text{proj}(\text{PrName}), \text{PrName}), \quad \text{worksFor}(\text{pers}(\text{SSN}), \text{proj}(\text{PrName}))\]

\[m_2: \quad \text{SELECT SSN, Salary} \quad \mapsto \quad \text{Employee}(\text{pers}(\text{SSN})), \quad \text{salary}(\text{pers}(\text{SSN}), \text{Salary})\]

Applying \(m_1\) and \(m_2\) to the database, generates a virtual data layer:

**Object terms:** \(\text{pers}(23AB), \text{proj}(\text{optique}), \ldots\)

**Values:** \(\text{optique}, 1500, \ldots\)

**ABox assertions:** \(\text{Employee}(\text{pers}(23AB)), \ldots\)

\(\text{Project}(\{\text{proj}(\text{optique})\}), \ldots\)

\(\text{projectName}(\text{proj}(\text{optique}), \text{optique}), \ldots\)

\(\text{worksFor}(\text{pers}(23AB), \text{proj}(\text{optique})), \ldots\)

\(\text{salary}(\text{pers}(23AB), 1500), \ldots\)
Semantics of mappings

To formally define the semantics of an OBDA instance \( \mathcal{J} = \langle \mathcal{P}, \mathcal{D} \rangle \), where \( \mathcal{P} = \langle \mathcal{O}, \mathcal{M}, \mathcal{S} \rangle \), we first need to define the semantics of mappings.

**Satisfaction of a mapping assertion with respect to a database**

An interpretation \( \mathcal{I} \) satisfies a mapping assertion \( \Phi(\vec{x}) \rightsquigarrow \Psi(\vec{x}) \) in \( \mathcal{M} \) with respect to a database \( \mathcal{D} \) for \( \mathcal{S} \), if the following FOL sentence is true in \( \mathcal{I} \cup \mathcal{D} \):

\[
\forall \vec{x}. \Phi(\vec{x}) \rightarrow \Psi(\vec{x})
\]

Intuitively, \( \mathcal{I} \) satisfies \( \Phi \rightsquigarrow \Psi \) w.r.t. \( \mathcal{D} \) if all facts obtained by evaluating \( \Phi \) over \( \mathcal{D} \) and then propagating the answers to \( \Psi \), hold in \( \mathcal{I} \).
Semantics of mappings – Example

$m_1$: SELECT SSN, PrName
FROM D_1  

$m_2$: SELECT SSN, Salary
FROM D_2, D_3

The following interpretation $I$ satisfies the mapping assertions $m_1$ and $m_2$ with respect to the above database:

$\Delta^I_O = \{\text{pers}(23AB), \text{proj}(\text{optique}), \ldots\}$,  
$\Delta^I_V = \{\text{optique}, 1500, \ldots\}$

$Employee^I = \{\text{pers}(23AB), \ldots\}$,  
$Project^I = \{\text{proj}(\text{optique}), \ldots\}$,  
$projectName^I = \{(\text{proj}(\text{optique}), \text{optique}), \ldots\}$,  
$worksFor^I = \{(\text{pers}(23AB), \text{proj}(\text{optique})), \ldots\}$,  
$salary^I = \{(\text{pers}(23AB), 1500), \ldots\}$
Let $I = (\Delta^I, \cdot^I)$ be an interpretation of the ontology $O$.

**Model of an OBDA instance**

$I$ is a **model** of $J = \langle P, D \rangle$, with $P = \langle O, M, S \rangle$ if:

- $I$ is a model of $O$, and
- $I$ satisfies $M$ w.r.t. $D$, i.e., it satisfies every assertion in $M$ w.r.t. $D$.

An OBDA instance $J$ is **satisfiable** if it admits at least one model.
Outline

1. Query rewriting wrt an OWL 2 QL ontology
2. Mapping specification
3. Saturation and optimization of the mapping
4. Query reformulation and optimization
Querying the OBDA system

**OBDA system** $\mathcal{K} = \langle \mathcal{T}, \mathcal{M}, \mathcal{D} \rangle$

- $DL$-$\text{Lite}_R$ TBox $\mathcal{T}$
- RDF graph $\mathcal{G}$ obtained from the mapping $\mathcal{M}$ and the data sources $\mathcal{D}$
- $\mathcal{G}$ can be viewed as the ABox

**Query answering**

- SPARQL query $q$ over $\mathcal{K}$
- If there is no existential restriction $B \sqsubseteq \exists R.C$ in $\mathcal{T}$, $q$ can be directly evaluated over $\mathcal{G}_{\text{sat}}$

**Saturated RDF graph** $\mathcal{G}_{\text{sat}}$

- Saturation of $\mathcal{G}$ w.r.t. $\mathcal{T}$
- H-complete ABox
How to handle the RDF graph $G_{\text{sat}}$ in practice?

**By materializing it**
- Materialization of $G$ (ETL) + saturation
  - Large volume
  - Maintenance
- Typical profile: OWL 2 RL

**By keeping it virtual**
- Query rewriting
  + No materialization required
- Saturated mapping $M_{\text{sat}}$
- Typical profile: OWL 2 QL

By Diego Calvanese (unibz)
H-complete ABox

[Rodriguez-Muro, Kontchakov, and Zakharyaschev 2013; Kontchakov and Zakharyaschev 2014]

ABox saturation

- H-complete ABox: contains all the inferable ABox assertions
- Let $\mathcal{K}$ be a $DL$-Lite$_R$ knowledge base, and let $\mathcal{K}_{sat}$ be the result of saturating $\mathcal{K}$. Then, for every ABox assertion $\alpha$, we have:

$$\mathcal{K} \models \alpha \iff \alpha \in \mathcal{K}_{sat}$$

Saturated mapping $\mathcal{M}_{sat}$ (also called $T$-mapping)

- Composition of the mapping $\mathcal{M}$ and the $DL$-Lite$_R$ TBox $\mathcal{T}$.
- $\mathcal{M}_{sat}$ applied to $\mathcal{D}$ produces $\mathcal{G}_{sat}$ (H-complete ABox).
- Does not depend of the SPARQL query $q$ (can be pre-computed).
- Can be optimized (exploiting query containment).
TBox, user-defined mapping assertions, and foreign key

\[ \text{Student} \sqcup \text{PostDoc} \sqcup \text{AssociateProfessor} \sqcup \exists \text{teaches} \sqsubseteq \text{Person} \]

\[ \text{Student}(\text{iri1}(\text{scode})) \iff \text{student}(\text{scode}, \text{fn}, \text{ln}) \quad (1) \]
\[ \text{PostDoc}(\text{iri2}(\text{acode})) \iff \text{academic}(\text{acode}, \text{fn}, \text{ln}, \text{pos}), \text{pos} = 9 \quad (2) \]
\[ \text{AssociateProfessor}(\text{iri2}(\text{acode})) \iff \text{academic}(\text{acode}, \text{fn}, \text{ln}, \text{pos}), \text{pos} = 2 \quad (3) \]
\[ \text{FacultyMember}(\text{iri2}(\text{acode})) \iff \text{academic}(\text{acode}, \text{fn}, \text{ln}, \text{pos}) \quad (4) \]
\[ \text{teaches}(\text{iri2}(\text{acode}), \text{iri3}(\text{course})) \iff \text{teaching}(\text{course}, \text{acode}) \quad (5) \]

FK: \[ \exists y_1. \text{teaching}(y_1, x) \rightarrow \exists y_2 y_3 y_4. \text{academic}(x, y_2, y_3, y_4) \]

By **saturating the mapping**, we obtain mapping assertions for Person

\[ \text{Person}(\text{iri1}(\text{scode})) \iff \text{student}(\text{scode}, \text{fn}, \text{ln}) \quad (6) \]
\[ \text{Person}(\text{iri2}(\text{acode})) \iff \text{academic}(\text{acode}, \text{fn}, \text{ln}, \text{pos}), \text{pos} = 9 \quad (7) \]
\[ \text{Person}(\text{iri2}(\text{acode})) \iff \text{academic}(\text{acode}, \text{fn}, \text{ln}, \text{pos}), \text{pos} = 2 \quad (8) \]
\[ \text{Person}(\text{iri2}(\text{acode})) \iff \text{academic}(\text{acode}, \text{fn}, \text{ln}, \text{pos}) \quad (9) \]
\[ \text{Person}(\text{iri2}(\text{acode})) \iff \text{teaching}(\text{course}, \text{acode}) \quad (10) \]

By **optimizing the mapping** using query containment and the FK, we can remove mapping assertions 7, 8, and 10

\[ \text{Person}(\text{iri1}(\text{scode})) \iff \text{student}(\text{scode}, \text{fn}, \text{ln}) \quad (6) \]
\[ \text{Person}(\text{iri2}(\text{acode})) \iff \text{academic}(\text{acode}, \text{fn}, \text{ln}, \text{pos}) \quad (9) \]
Outline

1. Query rewriting wrt an OWL 2 QL ontology
2. Mapping specification
3. Saturation and optimization of the mapping
4. Query reformulation and optimization
Query reformulation as implemented by Ontop

1. Tree-witness rewriting
   - Input: $q$ (SPARQL) and $\mathcal{T}$
   - Output: $q_{tw}$ (SPARQL)

2. Query unfolding
   - Input: $q_{tw}$ and $\mathcal{M}_{sat}$
   - Output: $q_{unf}$ (SQL)

3. Query optimization
   - Input: $q_{unf}$, primary and foreign keys
   - Output: $q_{opt}$ (SQL)
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 joining performance

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

Integrity constraints
- Primary and foreign keys, unique constraints
- Sometimes implicit
- Vital for query reformulation!
Reformulation example – 1. Unfolding

Saturated mapping

\[
\begin{align*}
Teacher(\text{iri2}(acode)) & \leftarrow \text{academic}(acode, fn, ln, pos), \ pos \in [1..8] \\
Teacher(\text{iri2}(acode)) & \leftarrow \text{teaching}(course, acode) \\
\text{firstName}(\text{iri1}(scode), fn) & \leftarrow \text{student}(scode, fn, ln) \\
\text{firstName}(\text{iri2}(acode), fn) & \leftarrow \text{academic}(acode, fn, ln, pos) \\
\text{lastName}(\text{iri1}(scode), ln) & \leftarrow \text{student}(scode, fn, ln) \\
\text{lastName}(\text{iri2}(acode), ln) & \leftarrow \text{academic}(acode, fn, ln, pos)
\end{align*}
\]

Query (we assume that the ontology is empty, hence \(q_{\text{tw}} = q\))

\[
q(x, y, z) \leftarrow Teacher(x), \text{firstName}(x, y), \text{lastName}(x, z)
\]

Query unfolding, and normalization, to make the join conditions explicit

\[
\begin{align*}
q_{\text{norm}}(x, y, z) & \leftarrow q_{\text{1unf}}(x), q_{\text{2unf}}(x_1, y), q_{\text{3unf}}(x_2, z), \ x = x_1, \ x = x_2 \\
q_{\text{1unf}}(\text{iri2}(acode)) & \leftarrow \text{academic}(acode, fn, ln, pos), \ pos \in [1..8] \\
q_{\text{1unf}}(\text{iri2}(acode)) & \leftarrow \text{teaching}(course, acode) \\
q_{\text{2unf}}(\text{iri1}(scode), fn) & \leftarrow \text{student}(scode, fn, ln) \\
q_{\text{2unf}}(\text{iri2}(acode), fn) & \leftarrow \text{academic}(acode, fn, ln, pos) \\
q_{\text{3unf}}(\text{iri1}(scode), ln) & \leftarrow \text{student}(scode, fn, ln) \\
q_{\text{3unf}}(\text{iri2}(acode), ln) & \leftarrow \text{academic}(acode, fn, ln, pos)
\end{align*}
\]
Reformulation example – 2. Structural optimization

Unfolded normalized query

\[ q_{\text{norm}}(x, y, z) \leftarrow q_{1\text{unf}}(x), \ q_{2\text{unf}}(x_1, y), \ q_{3\text{unf}}(x_2, z), \ x = x_1, \ x = x_2 \]

\[ q_{1\text{unf}}(\text{iri2}(a)) \leftarrow \text{academic}(a, f, l, p), \ p \in [1..8] \]

\[ q_{1\text{unf}}(\text{iri2}(a)) \leftarrow \text{teaching}(c, a) \]

\[ q_{2\text{unf}}(\text{iri1}(s), f) \leftarrow \text{student}(s, f, l) \]

\[ q_{2\text{unf}}(\text{iri2}(a), f) \leftarrow \text{academic}(a, f, l, p) \]

\[ q_{3\text{unf}}(\text{iri1}(s), l) \leftarrow \text{student}(s, f, l) \]

\[ q_{3\text{unf}}(\text{iri2}(a), l) \leftarrow \text{academic}(a, f, l, p) \]

- 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

\[ q_{\text{lift}}(\text{iri2}(a), y, z) \leftarrow \text{academic}(a, f_1, l_1, p_1), \]

\[ \text{student}(s, f_2, l_2), \]

\[ \text{student}(s_1, f_3, l_3), \]

\[ \text{iri2}(a) = \text{iri1}(s), \]

\[ \text{iri2}(a) = \text{iri1}(s_1), \]

\[ p_1 \in [1..8] \]

\[ q_{\text{lift}}(\text{iri2}(a), y, z) \leftarrow \text{academic}(a, f_1, l_1, p_1), \]

\[ \text{student}(s, f_2, l_2), \]

\[ \text{academic}(a_2, f_3, z, p_3), \]

\[ \text{iri2}(a) = \text{iri1}(s), \]

\[ \text{iri2}(a) = \text{iri2}(a_2), \]

\[ p_1 \in [1..8] \]

(One sub-query not shown)

\[ q_{\text{lift}}(\text{iri2}(a), y, z) \leftarrow \text{academic}(a, f_1, l_1, p_1), \]

\[ \text{academic}(a_1, y, l_2, p_2), \]

\[ \text{academic}(a_2, f_3, z, p_3), \]

\[ \text{iri2}(a) = \text{iri2}(a_1), \]

\[ \text{iri2}(a) = \text{iri2}(a_2), \]

\[ p_1 \in [1..8] \]
### Reformulation example – 2. Structural optimization

<table>
<thead>
<tr>
<th>Unfolded normalized query</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_{\text{norm}}(x, y, z) \leftarrow q_{\text{1unf}}(x), q_{\text{2unf}}(x_1, y), q_{\text{3unf}}(x_2, z), x = x_1, x = x_2 )</td>
</tr>
<tr>
<td>( q_{\text{1unf}}(\text{iri2}(a)) \leftarrow \text{academic}(a, f, l, p), p \in [1..8] )</td>
</tr>
<tr>
<td>( q_{\text{1unf}}(\text{iri2}(a)) \leftarrow \text{teaching}(c, a) )</td>
</tr>
<tr>
<td>( q_{\text{2unf}}(\text{iri1}(s), f) \leftarrow \text{student}(s, f, l) )</td>
</tr>
<tr>
<td>( q_{\text{2unf}}(\text{iri2}(a), f) \leftarrow \text{academic}(a, f, l, p) )</td>
</tr>
<tr>
<td>( q_{\text{3unf}}(\text{iri1}(s), l) \leftarrow \text{student}(s, f, l) )</td>
</tr>
<tr>
<td>( q_{\text{3unf}}(\text{iri2}(a), l) \leftarrow \text{academic}(a, f, l, p) )</td>
</tr>
</tbody>
</table>

- 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.

<table>
<thead>
<tr>
<th>Flattening (URI template lifting) – Part 2/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_{\text{lift}}(\text{iri2}(a), y, z) \leftarrow \text{teaching}(c, a), \text{student}(s, f_2, l_2), \text{student}(s_1, f_3, l_3), \text{iri2}(a) = \text{iri1}(s), \text{iri2}(a) = \text{iri1}(s_1) )</td>
</tr>
<tr>
<td>( q_{\text{lift}}(\text{iri2}(a), y, z) \leftarrow \text{teaching}(c, a), \text{student}(s, f_2, l_2), \text{academic}(a_2, f_3, z, p_3), \text{iri2}(a) = \text{iri1}(s), \text{iri2}(a) = \text{iri2}(a_2) )</td>
</tr>
<tr>
<td>( q_{\text{lift}}(\text{iri2}(a), y, z) \leftarrow \text{teaching}(c, a), \text{academic}(a_1, y, l_2, p_2), \text{student}(s, f_3, l_3), \text{iri2}(a) = \text{iri2}(a_1), \text{iri2}(a) = \text{iri1}(s) )</td>
</tr>
<tr>
<td>( q_{\text{lift}}(\text{iri2}(a), y, z) \leftarrow \text{teaching}(c, a), \text{academic}(a_1, y, l_2, p_2), \text{academic}(a_2, f_3, z, p_3), \text{iri2}(a) = \text{iri2}(a_1), \text{iri2}(a) = \text{iri2}(a_2) )</td>
</tr>
</tbody>
</table>
We are left with just two queries, that we can simplify by eliminating equalities

\[ q_{\text{struct}}(\text{iri2}(a), y, z) \leftarrow \text{academic}(a, f_1, l_1, p_1), \ p_1 \in [1..8], \]
\[ \text{academic}(a, y, l_2, p_2), \]
\[ \text{academic}(a, f_3, z, p_3) \]

\[ q_{\text{struct}}(\text{iri2}(a), y, z) \leftarrow \text{teaching}(c, a), \]
\[ \text{academic}(a, y, l_2, p_2), \]
\[ \text{academic}(a, f_3, z, p_3) \]

We can then exploit database constraints (such as primary keys) for semantic optimization of the query.

**Self-join elimination** (semantic optimization)

PK: \( \text{academic}(acode, f, l, p) \land \text{academic}(acode, f', l', p') \)
\[ \rightarrow (f = f') \land (l = l') \land (p = p') \]

\[ q_{\text{opt}}(\text{iri2}(a), y, z) \leftarrow \text{academic}(a, y, z, p_1), \ p_1 \in [1..8] \]
\[ q_{\text{opt}}(\text{iri2}(a), y, z) \leftarrow \text{teaching}(c, a), \text{academic}(a, y, z, p_2) \]
