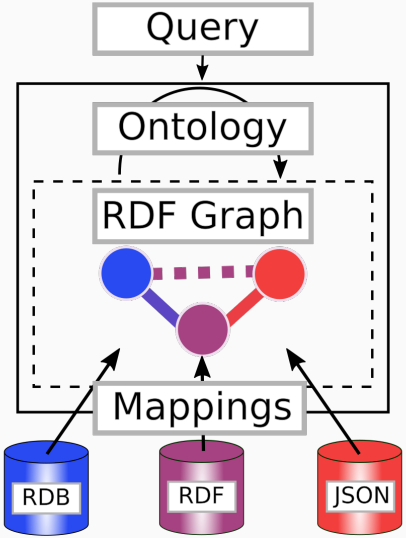
Obi-Wan: an RDF integration system

Maxime Buron

GraphIK - June 9, 2022

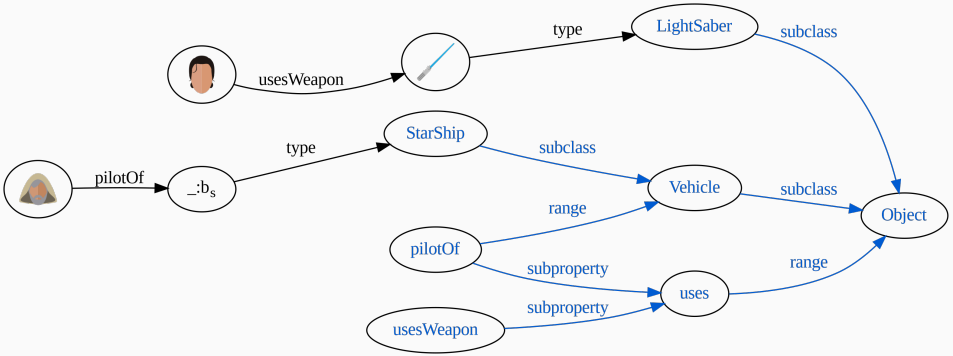
Ontology-Based Data Access (OBDA)



Outline

Preliminaries

RDF graph: data and RDFS ontology

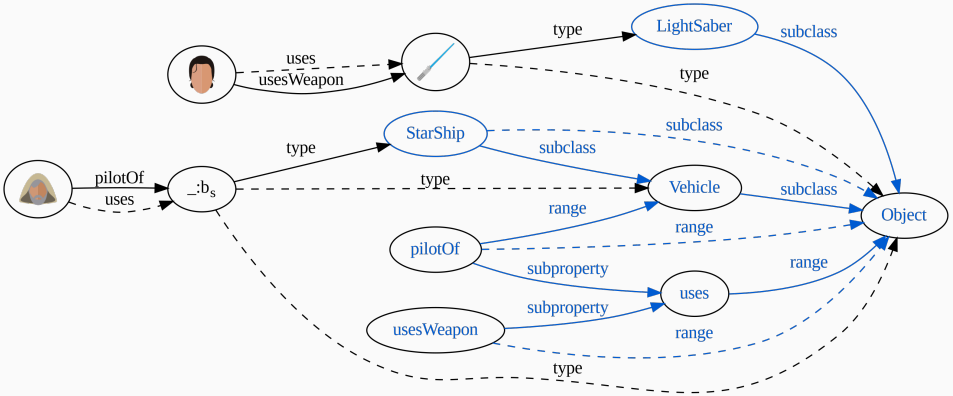


Graph saturation w.r.t. $\mathcal{R}_{\text{data}}$ and $\mathcal{R}_{\text{onto}}$

$$\mathcal{R}_{\text{data}} = \{ \quad (\mathrm{p}_a, :\mathit{subproperty}, \mathrm{p}_b), (\mathrm{s}, \mathrm{p}_a, \mathrm{o}) \rightarrow (\mathrm{s}, \mathrm{p}_b, \mathrm{o}) \quad \ldots \quad \}$$

$$\mathcal{R}_{\text{onto}} = \{ \quad (\mathrm{p}, :\mathit{subproperty}, \mathrm{p}_a), (\mathrm{p}_a, :\mathit{range}, \mathrm{o}) \rightarrow (\mathrm{p}, :\mathit{range}, \mathrm{o}) \quad \ldots \quad \}$$

The **full saturation** of G is $G^{\mathcal{R}_{\text{onto}} \cup \mathcal{R}_{\text{data}}}$:



Basic graph pattern queries

We consider queries over the data **and the ontology**.

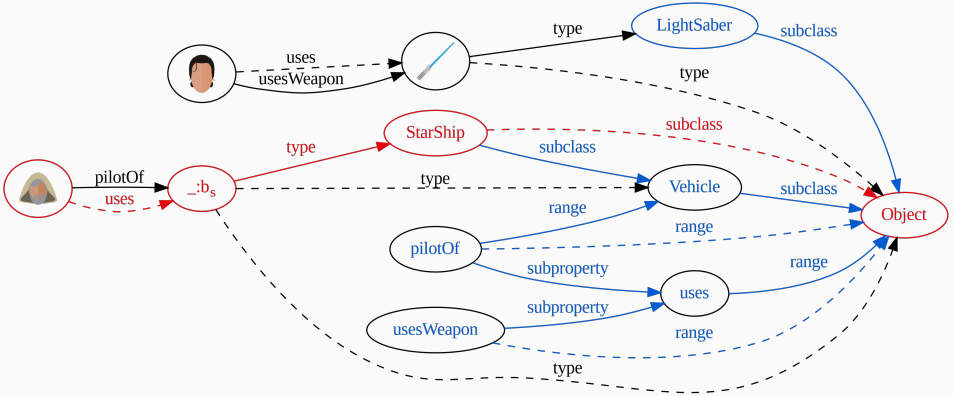
"Find anyone who uses something that is a kind of object"

$$q(x, y) \leftarrow (x, :uses, z), (z, :type, y), (y, :subclass, :Object)$$

Saturation-based query answering

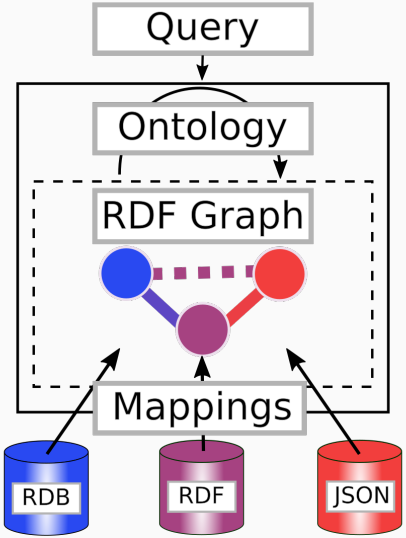
$$q(x, y) \leftarrow (x, :uses, z), (z, :type, y), (y, :subclass, :Object)$$

$$q(G^{\mathcal{R}_{\text{data}} \cup \mathcal{R}_{\text{onto}}}) = \langle \text{}, :LightSaber \rangle$$
$$\langle \text{  }, :Vehicle \rangle$$
$$\langle \text{  }, :StarShip \rangle$$



RDF integration system

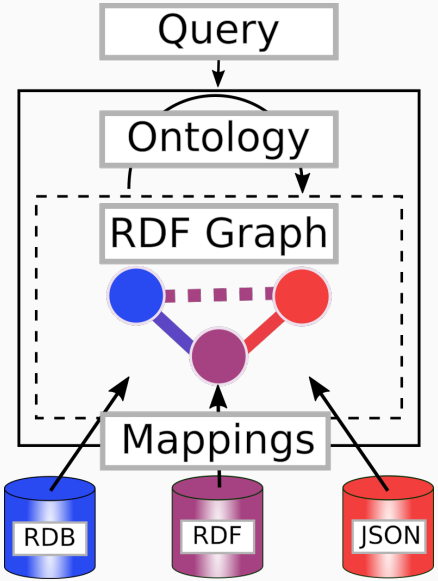
Ontology-Based Data Access



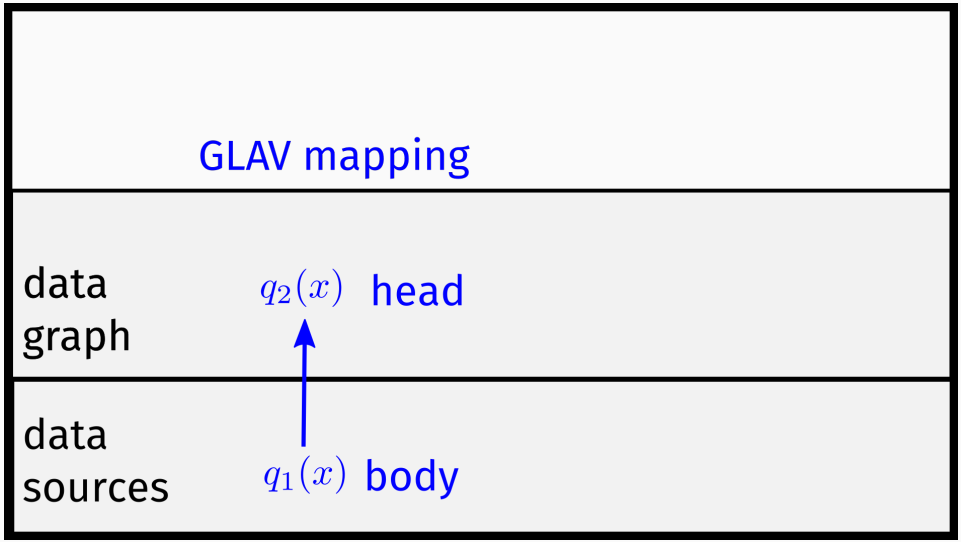
Contributions

More powerful integration setting:

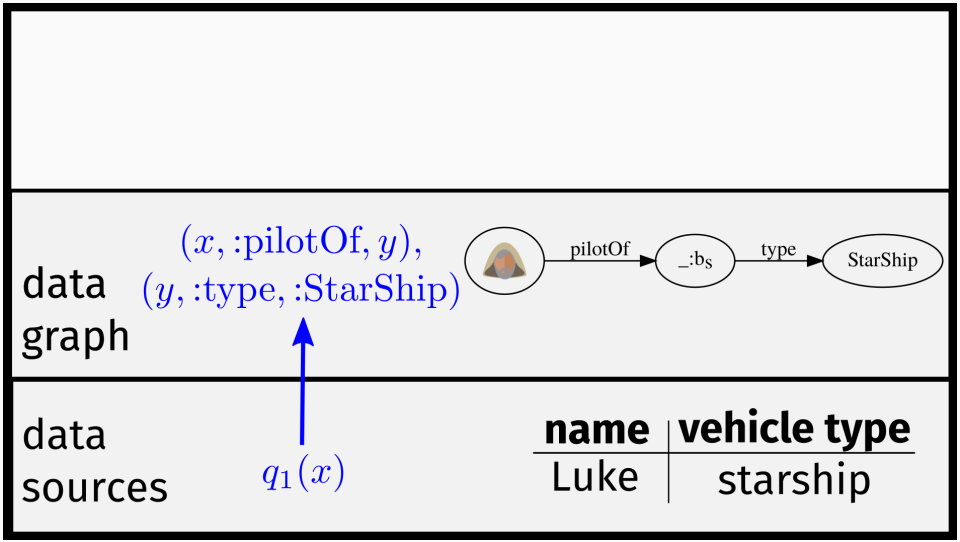
- Global-Local-As-View mappings in an OBDA context
- Queries on the data *and* the ontology



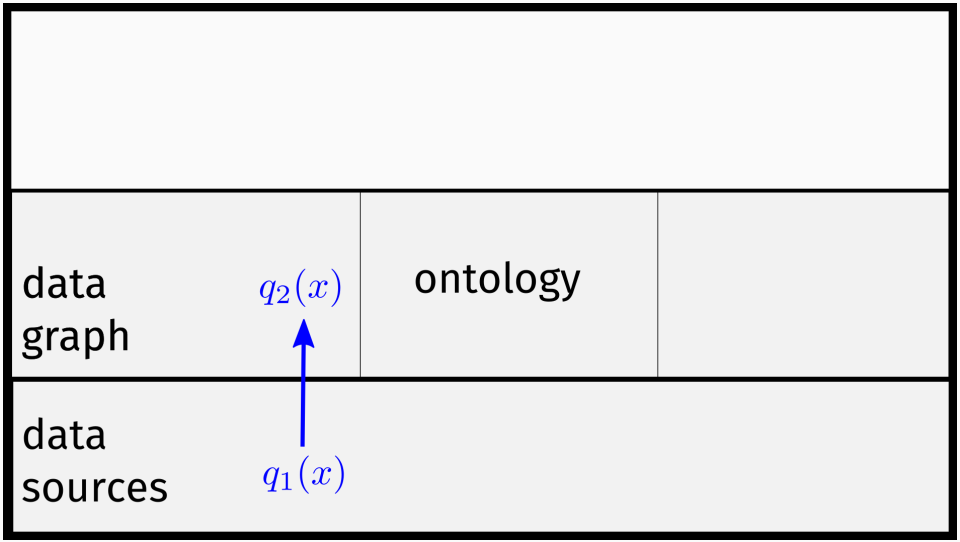
Global-Local-As-View mapping



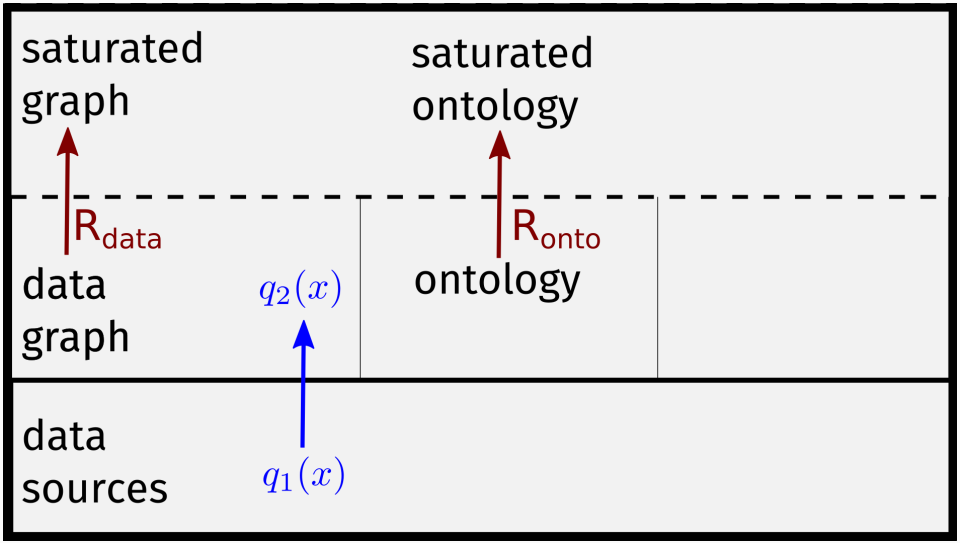
Global-Local-As-View mapping example



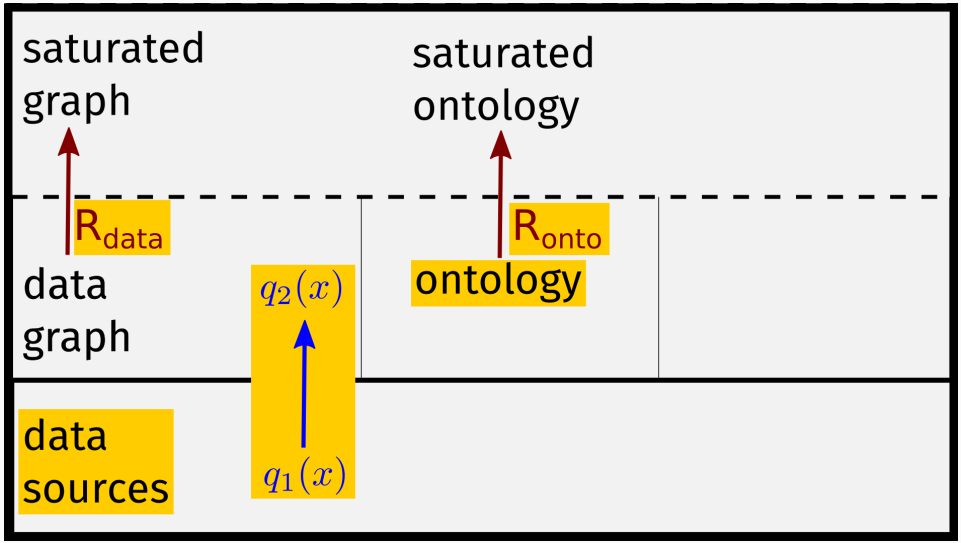
RDFS ontology



RDFS entailment



RDF Integration System

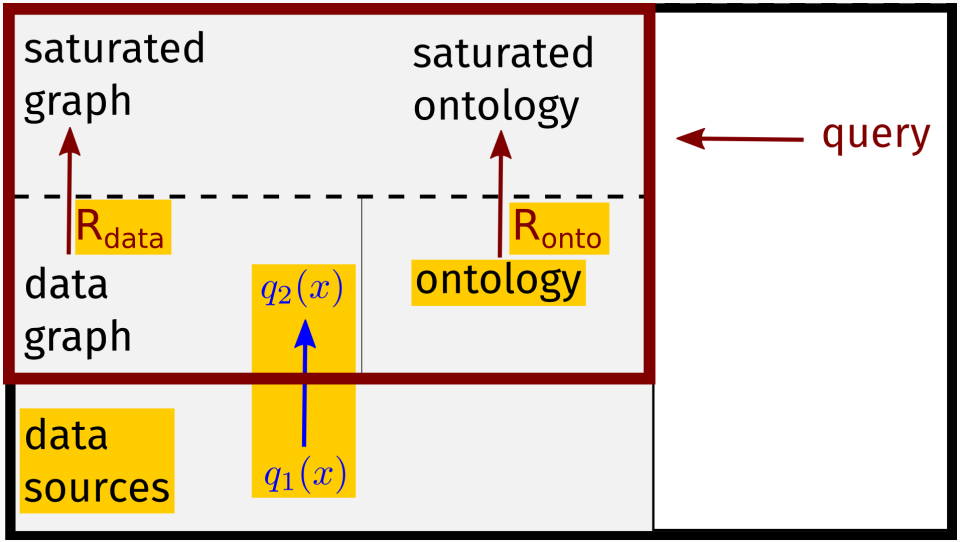


Obi-Wan

->Demonstration

`https://pages.saclay.inria.fr/maxime.buron/projects/obi-wan/app/ris/star-wars-example/index.html`

Query answering problem

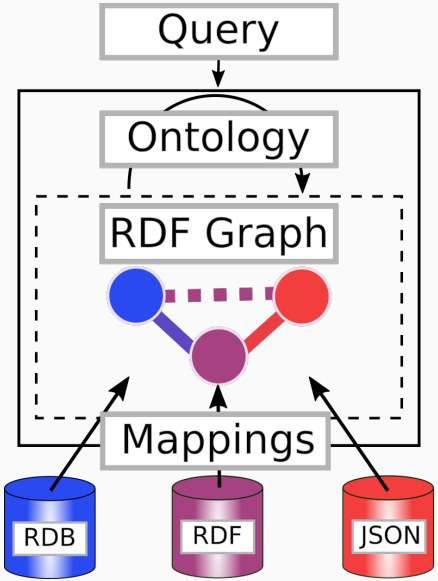


Query answering strategies

Materialization and mediation-based approaches

Obi-Wan dependencies for:

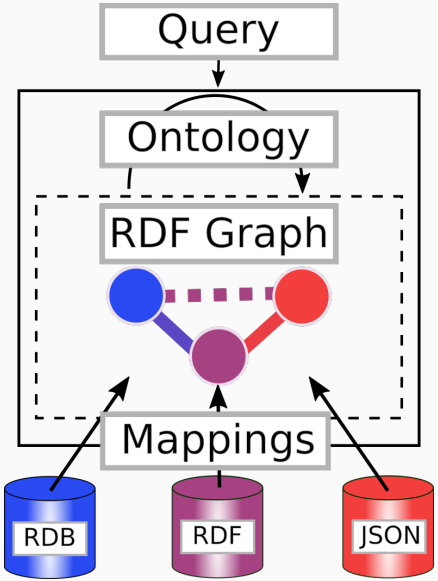
- materialization
  - OntoSQL (triple store)
- mediation
  - Graal (rewriting algorithm)
  - Tatooine (mediator)



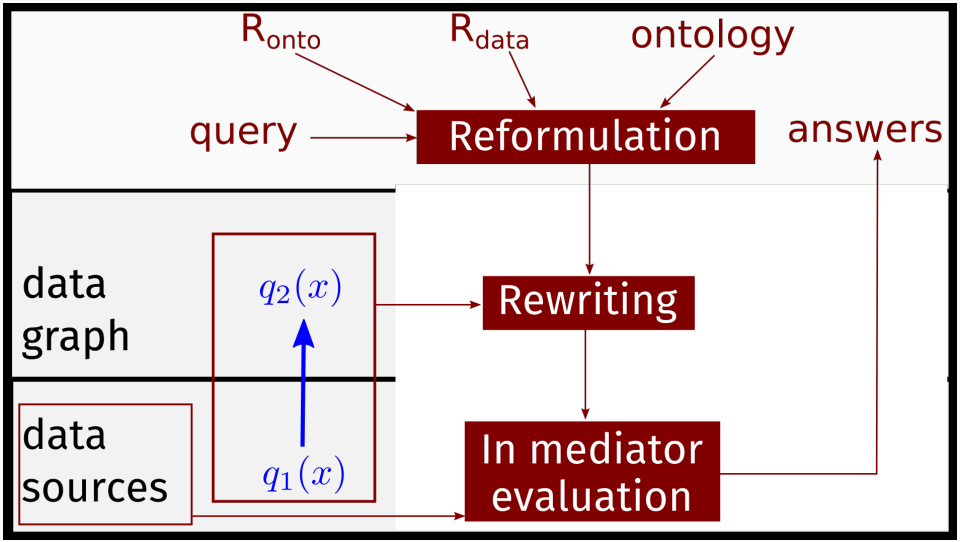
Obi-Wan query answering strategies

Obi-Wan implements:

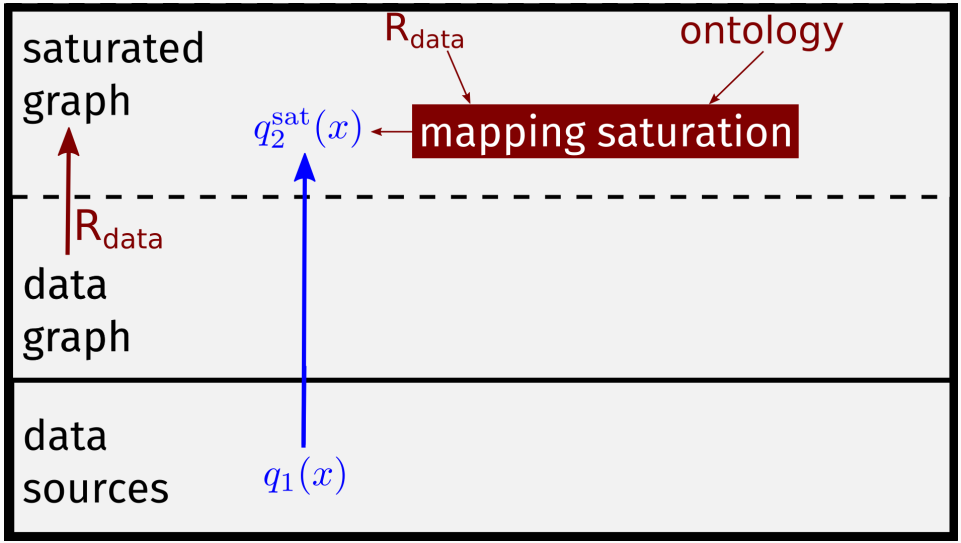
- several techniques to handle a part of the reasoning at *query time* or *offline*
- 9 query answering strategies based on materialization approach
- 4 query answering strategies based on mediation approach



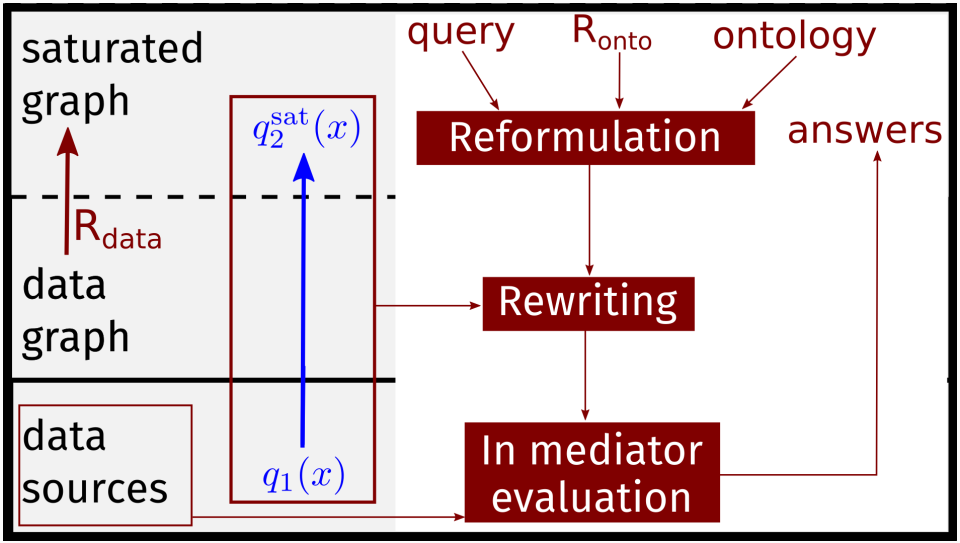
Classical strategy: *all* reasoning at query time (REW-CA)



Some reasoning at query time method (REW-C): mapping saturation



Some reasoning at query time method (REW-C): query time



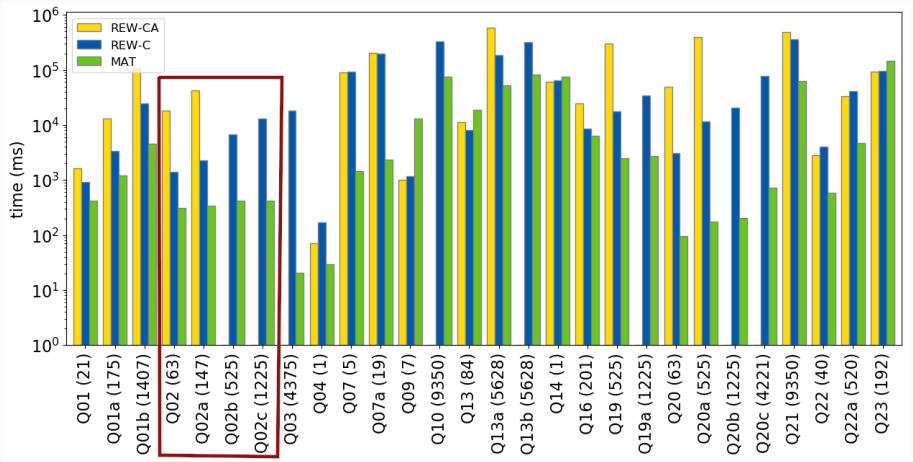
Experiments

RDF Integration System:

- Extension of Berlin SPARQL BenchMark
- 3863 GLAV mappings
- RDFS ontology of 2011 triples
- Induced graph with 108M triples (185M triples when saturated)
- Two data sources: One **relational** and one **JSON**

Query answering times on heterogeneous data sources

- Materialization (MAT) - kind of reference time
- Full reformulation + rewriting (REW-CA)
- Mapping saturation + partial reformulation + rewriting (REW-C)



Query rewriting using GLAV mappings

Global-Local As View mappings decomposition

Definition

Given m a GLAV mapping

$$q_1(\bar{x}) \rightarrow q_2(f_1(\bar{x}), \ldots, f_n(\bar{x}))$$

with $f_1, \ldots, f_n$ template functions.

We introduce V_m a view symbol to decompose m in two mappings:

1. mapping **lower part**

$$q_1(\bar{x}) \rightarrow V_m(f_1(\bar{x}), \ldots, f_n(\bar{x}))$$

2. mapping **upper part** or **LAV views**:

$$V_m(y_1, \ldots, y_n) \rightarrow q_2(y_1, \ldots, y_n)$$

Global-Local As View mappings decomposition

Definition

Given m a GLAV mapping

$$q_1(\bar{x}) \rightarrow q_2(f_1(\bar{x}), \ldots, f_n(\bar{x}))$$

with $f_1, \ldots, f_n$ template functions.

We introduce V_m a view symbol to decompose m in two mappings:

1. mapping **lower part**
$$q_1(\bar{x}) \rightarrow V_m(f_1(\bar{x}), \ldots, f_n(\bar{x}))$$
2. mapping **upper part** or **LAV views**:
$$V_m(y_1, \ldots, y_n) \rightarrow q_2(y_1, \ldots, y_n)$$

We want to rewriting a query on the integrated graph using the upper part of the mappings.

View-based query rewriting

Definition

A **view-based rewriting** of q is a sound and complete rewriting of q as a conjunctive query involving only views symbols.

We want to rewrite this query:

$$q() = (x, :type, :A), (x, :p, y), (y, :type, :B)$$

using these views :

- $V_1(u) \rightarrow (u, :type, :A)$
- $V_2(v) \rightarrow \exists u\ (u, :type, :A), (u, :p, v)$
- $V_3(w) \rightarrow (w, :type, :B)$

  In this example, we have a single complete rewriting:

$$\mathrm{rew}() = V_2(z), V_3(z)$$

Naive rewriting algorithm

1. compute the rewriting of the query using the views as existential rules
2. return the view-based rewriting

$$q() = (x, :type, :A), (x, :p, y), (y, :type, :B)$$

$V_1(u) \rightarrow (u, :type, :A)$

$V_2(v) \rightarrow \exists u\ (u, :type, :A), (u, :p, v)$

$V_3(w) \rightarrow (w, :type, :B)$

Naive rewriting algorithm

1. compute the rewriting of the query using the views as existential rules
2. return the view-based rewriting

$V_1(u) \rightarrow (u, :type, :A)$

$V_2(v) \rightarrow \exists u\ (u, :type, :A), (u, :p, v)$

$V_3(w) \rightarrow (w, :type, :B)$

$q() = (x, :type, :A), (x, :p, y), (y, :type, :B)$
$q_1() = V_1(x), (x, :p, y), (y, :type, :B)$

Naive rewriting algorithm

1. compute the rewriting of the query using the views as existential rules
2. return the view-based rewriting

$V_1(u) \rightarrow (u, :type, :A)$

$V_2(v) \rightarrow \exists u\ (u, :type, :A), (u, :p, v)$

$V_3(w) \rightarrow (w, :type, :B)$

$q() = (x, :type, :A), (x, :p, y), (y, :type, :B)$
$q_1() = V_1(x), (x, :p, y), (y, :type, :B)$
$q_2() = V_1(x), (x, :p, y), V_3(y)$

Naive rewriting algorithm

1. compute the rewriting of the query using the views as existential rules
2. return the view-based rewriting

$V_1(u) \rightarrow (u, :type, :A)$

$V_2(v) \rightarrow \exists u\ (u, :type, :A), (u, :p, v)$

$V_3(w) \rightarrow (w, :type, :B)$

$q() = (x, :type, :A), (x, :p, y), (y, :type, :B)$
$q_1() = V_1(x), (x, :p, y), (y, :type, :B)$
$q_2() = V_1(x), (x, :p, y), V_3(y)$
$q_3() = V_2(y), (y, :type, :B)$

Naive rewriting algorithm

1. compute the rewriting of the query using the views as existential rules
2. return the view-based rewriting

$V_1(u) \rightarrow (u, :type, :A)$

$V_2(v) \rightarrow \exists u\ (u, :type, :A), (u, :p, v)$

$V_3(w) \rightarrow (w, :type, :B)$

$q() = (x, :type, :A), (x, :p, y), (y, :type, :B)$
$q_1() = V_1(x), (x, :p, y), (y, :type, :B)$
$q_2() = V_1(x), (x, :p, y), V_3(y)$
$q_3() = V_2(y), (y, :type, :B)$
$\mathrm{rew}() = V_2(y), V_3(y)$

Naive rewriting algorithm

1. compute the rewriting of the query using the views as existential rules
2. return the view-based rewriting

$V_1(u) \rightarrow (u, :type, :A)$

$V_2(v) \rightarrow \exists u\ (u, :type, :A), (u, :p, v)$

$V_3(w) \rightarrow (w, :type, :B)$

$q() = (x, :type, :A), (x, :p, y), (y, :type, :B)$
$q_1() = V_1(x), (x, :p, y), (y, :type, :B)$
$q_2() = V_1(x), (x, :p, y), V_3(y)$
$q_3() = V_2(y), (y, :type, :B)$
$\mathrm{rew}() = V_2(y), V_3(y)$
$q_4() = (x, :type, :A), (x, :p, y), V_3(y)$

Naive rewriting algorithm

1. compute the rewriting of the query using the views as existential rules
2. return the view-based rewriting

$V_1(u) \rightarrow (u, :type, :A)$

$V_2(v) \rightarrow \exists u\ (u, :type, :A), (u, :p, v)$

$V_3(w) \rightarrow (w, :type, :B)$

$q() = (x, :type, :A), (x, :p, y), (y, :type, :B)$
$q_1() = V_1(x), (x, :p, y), (y, :type, :B)$
$q_2() = V_1(x), (x, :p, y), V_3(y)$
$q_3() = V_2(y), (y, :type, :B)$
$\mathrm{rew}() = V_2(y), V_3(y)$
$q_4() = (x, :type, :A), (x, :p, y), V_3(y)$
$q_2() = V_1(x), (x, :p, y), V_3(y)$

Naive rewriting algorithm

1. compute the rewriting of the query using the views as existential rules
2. return the view-based rewriting

$V_1(u) \rightarrow (u, :type, :A)$

$V_2(v) \rightarrow \exists u\ (u, :type, :A), (u, :p, v)$

$V_3(w) \rightarrow (w, :type, :B)$

$q() = (x, :type, :A), (x, :p, y), (y, :type, :B)$
$q_1() = V_1(x), (x, :p, y), (y, :type, :B)$
$q_2() = V_1(x), (x, :p, y), V_3(y)$
$q_3() = V_2(y), (y, :type, :B)$
$\mathrm{rew}() = V_2(y), V_3(y)$
$q_4() = (x, :type, :A), (x, :p, y), V_3(y)$
$q_2() = V_1(x), (x, :p, y), V_3(y)$
$\mathrm{rew}() = V_2(y), V_3(y)$

One step view-based rewriting algorithm

1. compute the single piece-unifiers of the query with the views
2. compute all sets of these unifiers that exactly cover the query
3. aggregate the unifiers contained in every set
4. rewrite the query with each aggregated unifiers

One step view-based rewriting algorithm

1. compute the single piece-unifiers of the query with the views
2. compute all sets of these unifiers that exactly cover the query
3. aggregate the unifiers contained in every set
4. rewrite the query with each aggregated unifiers

$V_1(u) \rightarrow (u, :type, :A)$

$V_2(v) \rightarrow \exists u\ (u, :type, :A), (u, :p, v)$

$V_3(w) \rightarrow (w, :type, :B)$

$q() = (x, :type, :A), (x, :p, y), (y, :type, :B)$

One step view-based rewriting algorithm

1. compute the single piece-unifiers of the query with the views
2. compute all sets of these unifiers that exactly cover the query
3. aggregate the unifiers contained in every set
4. rewrite the query with each aggregated unifiers

$V_1(u) \rightarrow (u, :type, :A)$

$V_2(v) \rightarrow \exists u\ (u, :type, :A), (u, :p, v)$

$V_3(w) \rightarrow (w, :type, :B)$

$q() = (x, :type, :A), (x, :p, y), (y, :type, :B)$

One step view-based rewriting algorithm

1. compute the single piece-unifiers of the query with the views
2. compute all sets of these unifiers that exactly cover the query
3. aggregate the unifiers contained in every set
4. rewrite the query with each aggregated unifiers

$V_1(u) \rightarrow (u, :type, :A)$

$V_2(v) \rightarrow \exists u\ (u, :type, :A), (u, :p, v)$

$V_3(w) \rightarrow (w, :type, :B)$

$q() = (x, :type, :A), (x, :p, y), (y, :type, :B)$

$\mathrm{rew}() = V_2(y), V_3(y)$

One step view-based rewriting algorithm

1. compute the single piece-unifiers of the query with the views
2. compute all sets of these unifiers that exactly cover the query
3. aggregate the unifiers contained in every set
4. rewrite the query with each aggregated unifiers

$V_1(u) \rightarrow (u, :type, :A)$

$V_2(v) \rightarrow \exists u\ (u, :type, :A), (u, :p, v)$

$V_3(w) \rightarrow (w, :type, :B)$

$q() = (x, :type, :A), (x, :p, y), (y, :type, :B)$
$\mathrm{rew}() = V_2(y), V_3(y)$

Exact cover

- NP-complete problem
- Knuth's algorithm X (Dancing Links technique for the implementation: DLX)

View-based rewriting optimizations

We **simplify** each view-based rewriting by:

1. computing its core
2. using the key on the views to simplify the rewriting

We **remove** every view-based rewriting, which:

1. are redundant, i.e., that are subsumed by another rewriting
2. contains a clash

Tatooine, the Obi-Wan mediator

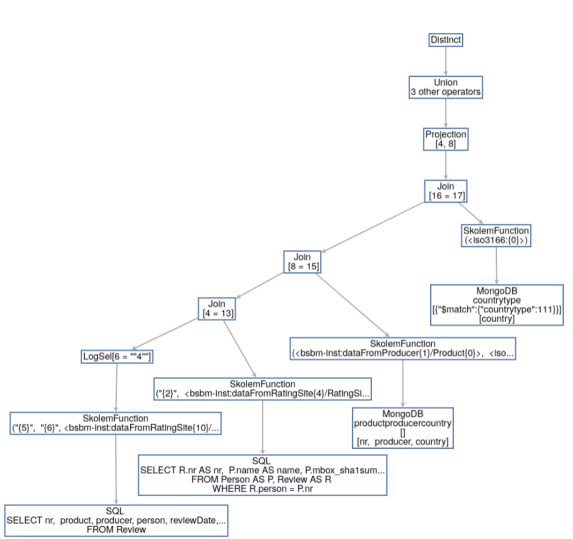
Tatooine overview

Tatooine

- wraps the answers from heterogeneous sources as tuples,
- allows to evaluate conjunctive query plans over heterogeneous sources.

Plan transformations before the evaluation:

1. original logical plan
2. optimized logical plan
3. physical plan



From view-based rewriting to original logical plan

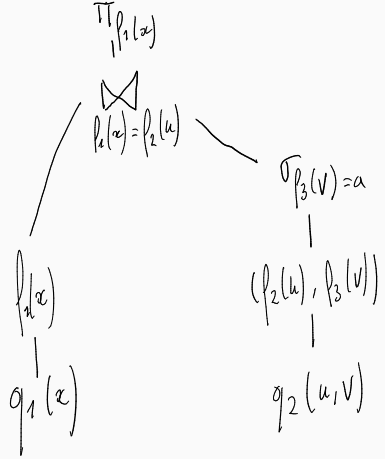
We unfold each view-based rewriting using the mapping lower parts to build a query plan.

A rewriting:

$$\mathrm{rew}(y) = V_1(y), V_2(y, a)$$

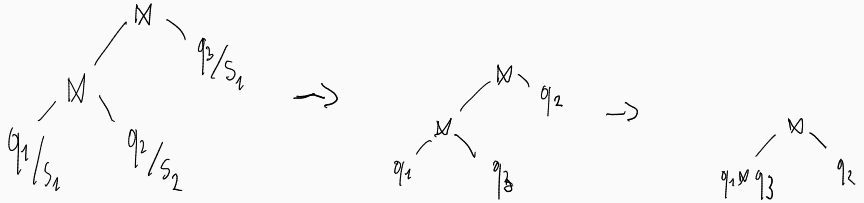
The mapping lower parts:

- $q_1(x) \rightarrow V_1(f_1(x))$
- $q_2(u, v) \rightarrow V_2(f_2(u), f_3(v))$



Plan optimizations

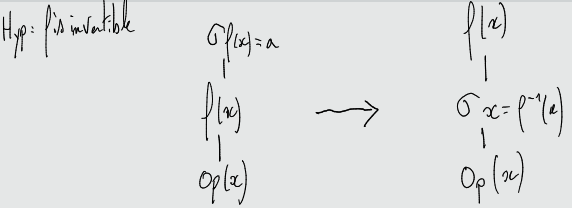
1. Tatooine optimizes query plan with:
   - local transformations of the plan
   - based on heuristics (without statistical information)
   - to push as many as possible operators to the source queries.
2. It also reorders the join operators in join trees to gather the ones that are on the same source.



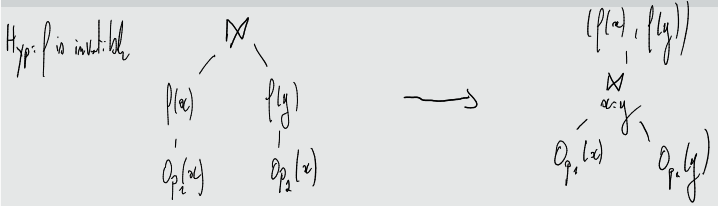
-> An optimized plan example

Local optimizations: moving up function operators

Pushing selection through function operator



Pushing join through function operators



From logical to physical plan : join implementations

- **Hash join**:
  1. loads the tuples from the left and right children
  2. performs the join using a hash of the values taken on the joint columns
- **Left (right) bind join**:
  1. reads the next tuple t from the left child
  2. selects the right tuples using the values taken by t on the joint columns
  3. concatenates t with each returned right tuples
  4. go to 1.

  Bind joins requires less memory, but needs the ability to select tuples from one child.

Conclusion

- Obi-Wan is an OBDA system that supports RDFS ontologies and GLAV mappings
- Obi-Wan proposes several materialization and mediation-based query answering strategies
- Query rewriting is performed using an one step view-based rewriting
- The mediator Tatooine implements several query plan optimizations
  source: `https://gitlab.inria.fr/cedar/obi-wan`