Mapping Management and Expressive Ontologies in Ontology-Based Data Access

4. Latest advancements in OBDA

Diego Calvanese, Benjamin Cogrel, Guohui Xiao

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

unibz
Freie Universität Bozen
Libera Università di Bolzano
Free University of Bozen-Bolzano

20th Int. Conf. on Knowledge Engineering and Knowledge Management (EKAW)
Bologna, 19 November 2016

Outline

unibz

Outline

unibz

Ontology-based data integration

[DBLP:conf/semweb/CalvaneseGHR15]

OBDI is a popular paradigm for integrating data sources:

- An ontology is connected to the data sources through **mappings**.
- The user can query a **virtual RDF graph** through SPARQL.
- The queries are translated into SQL queries over the data sources.

unibz

Ontology-based data integration

[DBLP:conf/semweb/CalvaneseGHR15]

OBDI is a popular paradigm for integrating data sources:

- An ontology is connected to the data sources through **mappings**.
- The user can query a **virtual RDF graph** through SPARQL.
- The queries are translated into SQL queries over the data sources.

Problem: information about one real-world entity can be distributed over several data sources.

1. Entity resolution: understand which records actually represent the same real world entity — We assume that this information is already available.
2. How to actually merge the data and provide a coherent view of it.

unibz

Merging data in OBDI

Physically merge the data (as done in ETL).

- Requires full control over the data sources.
- Requires to move the data $\rightsquigarrow$ issues with freshness, privacy, legal aspects.

$\rightsquigarrow$ Not possible in many real world scenarios!

Merging data in OBDI

Physically merge the data (as done in ETL).

- Requires full control over the data sources.
- Requires to move the data $\rightsquigarrow$ issues with freshness, privacy, legal aspects.

$\rightsquigarrow$ Not possible in many real world scenarios!

Use mappings to virtually merge the data: consistently generate only one URI per real world entity.

- Requires a central authority for defining URI schemas $\rightsquigarrow$ Does not scale well when data sources are added.
- For efficiency, URIs should be generated from the primary keys of the data sources, which in general differ.

Merging data in OBDI

Physically merge the data (as done in ETL).

- Requires full control over the data sources.
- Requires to move the data $\rightsquigarrow$ issues with freshness, privacy, legal aspects.

$\rightsquigarrow$ Not possible in many real world scenarios!

Use mappings to virtually merge the data: consistently generate only one URI per real world entity.

- Requires a central authority for defining URI schemas $\rightsquigarrow$ Does not scale well when data sources are added.
- For efficiency, URIs should be generated from the primary keys of the data sources, which in general differ.

None of these solutions is satisfactory!

Using owl:sameAs to link different datasources

- `owl:sameAs` (from now on `sameAs`) is a standard way of dealing with identity resolution in OWL (but not in OWL 2 QL)
  - E.g. `sameAs(:uni1/academic/3, :uni2/person/9)`
  - `sameAs` relation is an equivalence relation: reflexive, symmetric, and transitive.
- Challenges of using `sameAs` in OBDA
  1. Due to transitivity of `sameAs`, we lose query rewritability into SQL.
     $\leadsto$ Can we recover rewritability by restricting the linking mechanism?
  2. Similarly, for checking consistency of the data sources w.r.t. the ontology.
  3. Performance, to guarantee scalability over large enterprise datasets.

unibz

Dealing with transitivity of `sameAs` – Theoretical approach

We exploit partial materialization:

1. Expand the set $\mathcal{A}_S$ of `sameAs` facts into its reflexive, symmetric, and transitive closure $\mathcal{A}_S^*$.
   Note: we **do not** expand the data triples.

2. Transform a SPARQL query Q over $\langle \mathcal{T}, G \cup \mathcal{A}_S \rangle$ into $\varphi(Q)$ such that

$$cert(Q, \langle \mathcal{T}, G \cup \mathcal{A}_S \rangle) = cert(\varphi(Q), \langle \mathcal{T}, G \cup \mathcal{A}_S^* \rangle).$$

   The query $\varphi(Q)$ is obtained from Q by replacing every triple pattern t with $\varphi(t)$, where:

   - $\varphi(\{$`?v :P ?w`$\}) = \{$`?v sameAs _:a . _:a :P _:b . _:b sameAs ?w .`$\}$
   - $\varphi(\{$`?v rdf:type :C`$\}) = \{$`?v sameAs _:a . _:a rdf:type :C .`$\}$

unibz

Dealing with transitivity of `sameAs` – Theoretical approach

We exploit partial materialization:

1. Expand the set $\mathcal{A}_S$ of `sameAs` facts into its reflexive, symmetric, and transitive closure $\mathcal{A}_S^*$.
   Note: we **do not** expand the data triples.

2. Transform a SPARQL query Q over $\langle \mathcal{T}, G \cup \mathcal{A}_S \rangle$ into $\varphi(Q)$ such that

$$cert(Q, \langle \mathcal{T}, G \cup \mathcal{A}_S \rangle) = cert(\varphi(Q), \langle \mathcal{T}, G \cup \mathcal{A}_S^* \rangle).$$

   The query $\varphi(Q)$ is obtained from Q by replacing every triple pattern t with $\varphi(t)$, where:
   - $\varphi(\{$`?v :P ?w`$\}) = \{$`?v sameAs _:a . _:a :P _:b . _:b sameAs ?w .`$\}$
   - $\varphi(\{$`?v rdf:type :C`$\}) = \{$`?v sameAs _:a . _:a rdf:type :C .`$\}$

This approach is only theoretical, since:

- we are not given `sameAs` statements
- we want to avoid materializing all of $\mathcal{A}_S$ in the ontology.

unibz

Using sameAs Mapping in OBDA

Example sameAs mapping

- ```
  ex:uni1/academic/{a_id} owl:sameAs ex:uni2/person/{pid} .
  ← SELECT uni1.academic.a_id, uni2.person.pid
     FROM uni1.student, uni2.person
     WHERE uni1.student.ssn = uni2.person.ssn
  ```
- ```
  ex:uni1/academic/{a_id} owl:sameAs ex:uni2/person/{pid} .
  ← SELECT uni1.academic.a_id, uni2.person.pid
     FROM uni1.academic, uni2.person
     WHERE uni1.academic.ssn = uni2.person.ssn
  ```

Query reformulation using sameAs mapping

1. We assume that sameAs mappings already capture transitivity.
2. We add the symmetric version of each sameAs mapping assertion.
3. We deal with reflexivity by rewriting the user query.

Example

Query

SELECT ?p ?fn ?ln WHERE {
?p a foaf:Person .
?p :first_name ?fn .
?p :last_name ?ln .
}

Answer

p	fn	ln
:uni1/academic/3	Rachel	Ward
:uni2/person/9	Rachel	Ward
:uni1/academic/11	Alvena	Merry
:uni1/student/20	Alvena	Merry
:uni2/person/3	Alvena	Merry
...		

Limitations with `owl:sameAs`

Inherent issues with `owl:sameAs`

- Performance issue
  - The size of $\phi(Q)$ w.r.t. `sameAs` is exponentially larger than Q in general
  - Expensive to execute
- Repeated semantically equivalent results
  - Difficult to understand due to semantically duplicates

Canonical IRI as a rescue

- Break the symmetry!
- Each entity may has several IRIs, but only **a single canonical representation**.

Canonical IRI assertions

We assume that $\mathcal{G}$ is augmented with a set $\mathcal{A}_C$ of *canonical IRI assertions* using the property `canIriOf`.

Assumption on $\mathcal{A}_C$

- `canIriOf` is inverse functional in $\mathcal{A}_C$:
  $\{\ \mathtt{canIriOf}(c_1, i),\ \mathtt{canIriOf}(c_2, i)\ \} \subseteq \mathcal{A}_C$ implies $c_1 = c_2$.
- $\mathtt{canIriOf} \sqsubseteq \mathtt{sameAs}$

Example canonical IRI assertions

- `canIriOf(:person/ward-987183, :uni1/academic/3)`
- `canIriOf(:person/ward-987183, :uni2/person/9)`

unibz

Query answering under canonical IRI semantics

Canonical IRI/graph function

- Canonical IRI function:

$$can_{\mathcal{A}_C}(i) = \begin{cases} c_i, & \text{if} \texttt{canIriOf}(c_i, i) \in \mathcal{A}_C \\ i, & \text{otherwise} \end{cases}$$

- Canonical graph function:

$$\begin{aligned} can_{\mathcal{A}_C}(\mathcal{G}) = & \{A(can_{\mathcal{A}_C}(i)) \mid A(i) \in \mathcal{G}\} \\ & \cup \{P(can_{\mathcal{A}_C}(i), can_{\mathcal{A}_C}(i)\} \mid P(i,j) \in \mathcal{G}\} \end{aligned}$$

Query answering under canonical IRI semantics

$$cert\_can(Q, \langle \mathcal{T}, \mathcal{G} \cup \mathcal{A}_C \rangle) = cert(Q, \langle \mathcal{T}, can_{\mathcal{A}_C}(\mathcal{G}^{sat}) \rangle).$$

where $\mathcal{G}^{sat}$ is the saturated ABox of $\langle \mathcal{T}, \mathcal{G} \rangle$.

unibz

Canonical IRI semantics in OBDA

Example canIriOf mapping

- ex:person/{ssn} canIriOf ex:uni1/academic/{a_id} .
  ← SELECT uni1.academic.a_id, uni1.academic.ssn
  FROM uni1.academic
- ex:person/{ssn} canIriOf ex:uni1/student/{s_id} .
  ← SELECT uni1.student.s_id, uni1.student.ssn
  FROM uni1.student
- ex:person/{ssn} canIriOf ex:uni2/person/{pid} .
  ← SELECT uni2.person.pid, uni2.person.ssn
  FROM uni2.person

Query reformulation using canIriOf mapping

1. We developed a mapping rewriting algorithm.

unibz

Example under canonical IRI semantics

Query

SELECT ?p ?fn ?ln WHERE {
?p a foaf:Person .
?p :first_name ?fn .
?p :last_name ?ln .
}

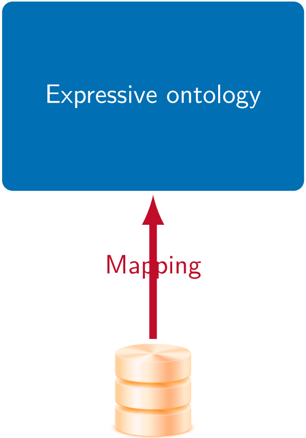
Answer

p	fn	ln
:person/ward-987183	Rachel	Ward
:person/merry-98821	Alvena	Merry
...		

Outline

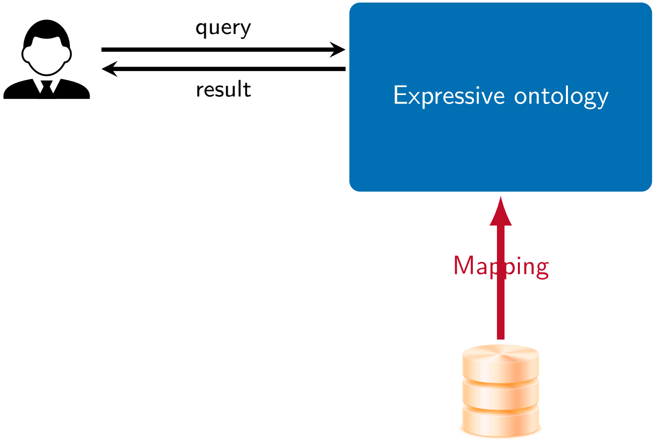
Requirement: deal with more expressive ontologies

[2016-aaai-ontoprox]



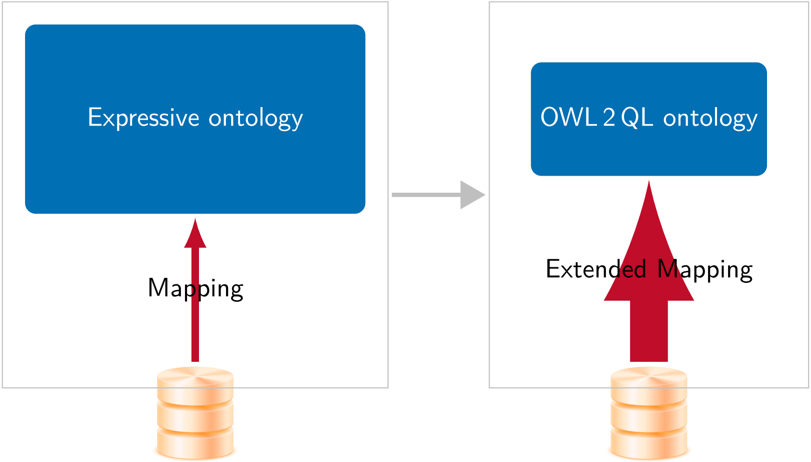
Requirement: deal with more expressive ontologies

[2016-aaai-ontoprox]



Rewritings the OBDA specification

We exploit the expressivity of the mapping layer, to compile ontology knowledge into the mapping.



Example: compiling ontology constraints into the mapping

Example

$$\mathcal{T} = \{\ A \sqcap B \sqsubseteq C\ \}$$
$$\mathcal{M} = \{\ \mathsf{SQL}_A(x) \rightsquigarrow A(x),$$
$$\mathsf{SQL}_B(x) \rightsquigarrow B(x)\ \}$$

unibz

Example: compiling ontology constraints into the mapping

Example

$$\begin{aligned} \mathcal{T} &= \{\ A \sqcap B \sqsubseteq C\ \} \\ \mathcal{M} &= \{\ \mathsf{SQL}_A(x) \rightsquigarrow A(x), \\ &\qquad \mathsf{SQL}_B(x) \rightsquigarrow B(x)\ \} \end{aligned} \quad \Rightarrow \quad \begin{aligned} \mathcal{T}' &= \{\ \ \} \\ \mathcal{M}' &= \{\ \mathsf{SQL}_A(x) \rightsquigarrow A(x), \\ &\qquad \mathsf{SQL}_B(x) \rightsquigarrow B(x), \\ &\qquad \mathsf{SQL}_A(x) \land \mathsf{SQL}_B(x) \rightsquigarrow C(x)\ \} \end{aligned}$$

Example: compiling ontology constraints into the mapping

Example

$$\begin{aligned}\mathcal{T} &= \{\ A \sqcap B \sqsubseteq C\ \} \\ \mathcal{M} &= \{\ \mathsf{SQL}_A(x) \rightsquigarrow A(x), \\ &\quad\ \ \mathsf{SQL}_B(x) \rightsquigarrow B(x)\ \}\end{aligned} \quad\Rightarrow\quad \begin{aligned}\mathcal{T}' &= \{\ \ \} \\ \mathcal{M}' &= \{\ \mathsf{SQL}_A(x) \rightsquigarrow A(x), \\ &\quad\ \ \mathsf{SQL}_B(x) \rightsquigarrow B(x), \\ &\quad\ \ \mathsf{SQL}_A(x) \land \mathsf{SQL}_B(x) \rightsquigarrow C(x)\ \}\end{aligned}$$

Example

$$\begin{aligned}\mathcal{T} &= \{\ \exists R.A \sqsubseteq C\ \} \\ \mathcal{M} &= \{\ \mathsf{SQL}_A(x) \rightsquigarrow A(x), \\ &\quad\ \ \mathsf{SQL}_R(x,y) \rightsquigarrow R(x,y)\ \}\end{aligned}$$

Example: compiling ontology constraints into the mapping

Example

$$
\begin{array}{l}
\mathcal{T} = \{\ A \sqcap B \sqsubseteq C\ \} \\
\mathcal{M} = \{\ \mathsf{SQL}_A(x) \rightsquigarrow A(x), \\
\qquad\quad \mathsf{SQL}_B(x) \rightsquigarrow B(x)\ \}
\end{array}
\Rightarrow
\begin{array}{l}
\mathcal{T}' = \{\ \ \} \\
\mathcal{M}' = \{\ \mathsf{SQL}_A(x) \rightsquigarrow A(x), \\
\qquad\quad \mathsf{SQL}_B(x) \rightsquigarrow B(x), \\
\qquad\quad \mathsf{SQL}_A(x) \land \mathsf{SQL}_B(x) \rightsquigarrow C(x)\ \}
\end{array}
$$

Example

$$
\begin{array}{l}
\mathcal{T} = \{\ \exists R.A \sqsubseteq C\ \} \\
\mathcal{M} = \{\ \mathsf{SQL}_A(x) \rightsquigarrow A(x), \\
\qquad\quad \mathsf{SQL}_R(x,y) \rightsquigarrow R(x,y)\ \}
\end{array}
\Rightarrow
\begin{array}{l}
\mathcal{T}' = \{\ \ \} \\
\mathcal{M}' = \{\ \mathsf{SQL}_A(x) \rightsquigarrow A(x), \\
\qquad\quad \mathsf{SQL}_R(x,y) \rightsquigarrow R(x,y), \\
\qquad\quad \mathsf{SQL}_R(x,y) \land \mathsf{SQL}_A(y) \rightsquigarrow C(x)\ \}
\end{array}
$$

Example: Recursion

Recursion cannot be fully captured via the mapping.
$\rightsquigarrow$ We use approximation, by setting a bound on the depth of the Datalog expansion of queries.

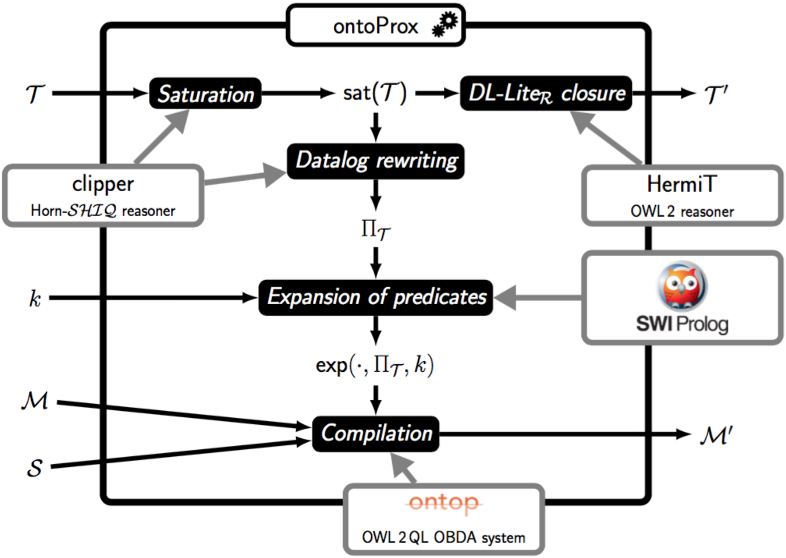
Example

$$
\begin{aligned}
\mathcal{T} &= \{\ \exists R.A \sqsubseteq A\ \} \\
\mathcal{M} &= \{\ \mathsf{SQL}_A(x) \rightsquigarrow A(x), \\
&\qquad \mathsf{SQL}_R(x,y) \rightsquigarrow R(x,y)\ \}
\end{aligned}
$$

$\Rightarrow$

$$
\begin{aligned}
\mathcal{T}' &= \{\ \ \} \\
\mathcal{M}' &= \{\ \mathsf{SQL}_A(x) \rightsquigarrow A(x), \\
&\qquad \mathsf{SQL}_R(x,y) \rightsquigarrow R(x,y), \\
&\qquad \mathsf{SQL}_R(x,y) \land \mathsf{SQL}_A(y) \rightsquigarrow A(x) \\
&\qquad \mathsf{SQL}_R(x,y) \land \mathsf{SQL}_R(y,z) \land \mathsf{SQL}_A(z) \rightsquigarrow A(x) \\
&\qquad \mathsf{SQL}_R(x,y) \land \mathsf{SQL}_R(y,z) \land \mathsf{SQL}_R(z,w) \land \mathsf{SQL}_A(w) \rightsquigarrow A(x) \\
&\qquad \ldots\ \}
\end{aligned}
$$

Implementation



unibz

Key points

- Framework for the Rewriting/Approximation of OBDA specifications by exploiting mappings
- Integration of existing techniques:
  - Datalog rewritability of expressive DLs (e.g., Horn-$\mathcal{ALCHIQ}$),
  - boundedness of Datalog programs
  - first-order rewritability of expressive DLs
- A novel technique to capture the anonymous part of the canonical models of the original TBox by an OWL 2 QL TBox.
- Ongoing work: implementation and benchmark

unibz

Outline

Example: Non-relational Database MongoDB

MongoDB is a popular database storing collections of JSON-like documents:

```
{ "_id": 4,
  "awards": [ {"award": "Rosing Prize", "year": 1999},
              {"award": "Turing Award", "by": "ACM", "year": 2001},
              {"award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE"} ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": {"first": "Kristen", "last": "Nygaard"}
}
```

Example: Non-relational Database MongoDB

MongoDB is a popular database storing collections of JSON-like documents:

Keys

```
{ "_id": 4,
  "awards": [ {"award": "Rosing Prize", "year": 1999},
              {"award": "Turing Award", "by": "ACM", "year": 2001},
              {"award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE"} ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": {"first": "Kristen", "last": "Nygaard"}
}
```

Example: Non-relational Database MongoDB

MongoDB is a popular database storing collections of JSON-like documents:

Values

```
{ "_id": 4,
  "awards": [ {"award": "Rosing Prize", "year": 1999},
              {"award": "Turing Award", "by": "ACM", "year": 2001},
              {"award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE"} ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": {"first": "Kristen", "last": "Nygaard"}
}
```

Example: Non-relational Database MongoDB

MongoDB is a popular database storing collections of JSON-like documents:

Arrays

```
{ "_id": 4,
  "awards": [ {"award": "Rosing Prize", "year": 1999},
              {"award": "Turing Award", "by": "ACM", "year": 2001},
              {"award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE"} ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": {"first": "Kristen", "last": "Nygaard"}
}
```

Example: Non-relational Database MongoDB

MongoDB is a popular database storing collections of JSON-like documents:

```
{ "_id": 4,
  "awards": [ {"award": "Rosing Prize", "year": 1999},
              {"award": "Turing Award", "by": "ACM", "year": 2001},
              {"award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE"} ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": {"first": "Kristen", "last": "Nygaard"}
}
```

Nested Objects

Example: Non-relational Database MongoDB

MongoDB is a popular database storing collections of JSON-like documents:

```
{ "_id": 4,
  "awards": [ {"award": "Rosing Prize", "year": 1999},
              {"award": "Turing Award", "by": "ACM", "year": 2001},
              {"award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE"} ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": {"first": "Kristen", "last": "Nygaard"}
}
```

Example: Non-relational Database MongoDB

MongoDB is a popular database storing collections of JSON-like documents:

```
{ "_id": 4,
  "awards": [ {"award": "Rosing Prize", "year": 1999},
              {"award": "Turing Award", "by": "ACM", "year": 2001},
              {"award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE"} ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": {"first": "Kristen", "last": "Nygaard"}
}
```

MongoDB provides powerful, but unconventional query capabilities. The query below retrieves from the `bios` collection persons who received two awards in the same year:

```
db.bios.aggregate([
  {$project : {"name": true, "award1": "$awards", "award2": "$awards" }},
  {$unwind: "$award1"}, {$unwind: "$award2"},
  {$project: {"name": true, "award1": true, "award2": true,
             "twoInOneYear": { $and: [ {$eq: ["$award1.year", "$award2.year"]},
                                      {$ne: ["$award1.award", "$award2.award"]} ]}}},
  {$match: {"twoInOneYear": true} },
  {$project : {"firstName": "$name.first",   "lastName": "$name.last" ,
              "awardName1": "$award1.award", "awardName2": "$award2.award",
              "year": "$award1.year" }}
])
```

Example: Non-relational Database MongoDB

MongoDB is a popular database storing collections of JSON-like documents:

```
{ "_id": 4,
  "awards": [ {"award": "Rosing Prize", "year": 1999},
              {"award": "Turing Award", "by": "ACM", "year": 2001},
              {"award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE"} ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": {"first": "Kristen", "last": "Nygaard"}
}
```

MongoDB provides powerful, but unconventional query capabilities. The query below retrieves from the bios collection persons who received two awards in the same year:

```
db.bios.aggregate([
  {$project : {"name": true, "award1": "$awards", "award2": "$awards" }},
  {$unwind: "$award1"}, {$unwind: "$award2"},
  {$project: {"name": true, "award1": true, "award2": true,
           "twoInOneYear": { $and: [ {$eq: ["$award1.year", "$award2.year"]},
                                     {$ne: ["$award1.award", "$award2.award"]} ]}}},
  {$match: {"twoInOneYear": true} },
  {$project : {"firstName": "$name.first",    "lastName": "$name.last" ,
            "awardName1": "$award1.award", "awardName2": "$award2.award",
            "year": "$award1.year" }}
])
```

Paths: concatenations of keys

Example: Non-relational Database MongoDB

MongoDB is a popular database storing collections of JSON-like documents:

```
{ "_id": 4,
  "awards": [ {"award": "Rosing Prize", "year": 1999},
              {"award": "Turing Award", "by": "ACM", "year": 2001},
              {"award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE"} ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": {"first": "Kristen", "last": "Nygaard"}
}
```

MongoDB provides powerful, but unconventional query capabilities. The query below retrieves from the bios collection persons who received two awards in the same year:

```
db.bios.aggregate([
  {$project : {"name": true
  {$unwind: "$award1"}, {$u
  {$project: {"name": true,
            "twoInOneYear"

  {$match: {"twoInOneYear":
  {$project : {"firstName": "$name.first",    "lastName": "$name.last" ,
             "awardName1": "$award1.award", "awardName2": "$award2.award",
             "year": "$award1.year" }}
])
```

This is a MUP (match-unwind-project) query performing an inner-document join.

In [BCCRX16] we show that MUPG (match-uniwind-project-group) queries capture full Relational Algebra over a single collection.

JSON-RDF mapping example [2016-dl-obda-mongo]

```
{ "_id": 4,
  "awards": [ {"award": "Rosing Prize", "year": 1999},
              {"award": "Turing Award", "by": "ACM", "year": 2001},
              {"award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE"} ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": {"first": "Kristen", "last": "Nygaard"} }
```

JSON-RDF mapping example [2016-dl-obda-mongo]

D

```
{ "_id": 4,
  "awards": [ {"award": "Rosing Prize", "year": 1999},
              {"award": "Turing Award", "by": "ACM", "year": 2001},
              {"award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE"} ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": {"first": "Kristen", "last": "Nygaard"} }
```

M

$\mathcal{M}$: $q_s \rightsquigarrow_{\mathbf{K}}$ (?X a :*Scientist*) .
(?X :*firstName* ?F) .
(?X :*lastName* ?L) .
(?X :*gotAward* ?A) .
(?A :*awardedInYear* ?Y) .
(?A :*awardName* ?N)

Retrieves all documents from the `bios` collection

JSON-RDF mapping example [2016-dl-obda-mongo]

D


```
{ "_id": 4,
  "awards": [ {"award": "Rosing Prize", "year": 1999},
              {"award": "Turing Award", "by": "ACM", "year": 2001},
              {"award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE"} ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": {"first": "Kristen", "last": "Nygaard"} }
```


Variable to term maps

M

$$\begin{aligned}\mathbf{K} = \{\ & ?X \mapsto \texttt{:/\{\_id\}}, \\ & ?F \mapsto \texttt{\{name.first\}}, \\ & ?L \mapsto \texttt{\{name.last\}}, \\ & ?A \mapsto \texttt{:/\{\_id\}/}\mathsf{Award}\texttt{/\{awards.\#\}}, \\ & ?Y \mapsto \texttt{\{awards.\#.year\}}, \\ & ?N \mapsto \texttt{\{awards.\#.award\}}\ \}.\end{aligned}$$

$\mathcal{M}$: $q_s \rightsquigarrow_{\mathbf{K}}$ (?X a :*Scientist*) .
(?X :*firstName* ?F) .
(?X :*lastName* ?L) .
(?X :*gotAward* ?A) .
(?A :*awardedInYear* ?Y) .
(?A :*awardName* ?N)

Retrieves all documents from the `bios` collection

unibz

JSON-RDF mapping example [2016-dl-obda-mongo]

D

```
{ "_id": 4,
  "awards": [ {"award": "Rosing Prize", "year": 1999},
              {"award": "Turing Award", "by": "ACM", "year": 2001},
              {"award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE"} ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": {"first": "Kristen", "last": "Nygaard"} }
```

Variable to term maps

M

$$
\mathbf{K} = \{ \; ?X \mapsto :/\{\texttt{\_id}\}, \;
?F \mapsto \{\texttt{name.first}\}, \;
?L \mapsto \{\texttt{name.last}\}, \;
?A \mapsto :/\{\texttt{\_id}\}/\text{Award}/\{\texttt{awards.\#}\}, \;
?Y \mapsto \{\texttt{awards.\#.year}\}, \;
?N \mapsto \{\texttt{awards.\#.award}\} \; \}.
$$

$\mathcal{M}$: $q_s \rightsquigarrow_{\mathbf{K}}$ $(?X \text{ a } :\textit{Scientist})$. $(?X :\textit{firstName} \; ?F)$. $(?X :\textit{lastName} \; ?L)$. $(?X :\textit{gotAward} \; ?A)$. $(?A :\textit{awardedInYear} \; ?Y)$. $(?A :\textit{awardName} \; ?N)$

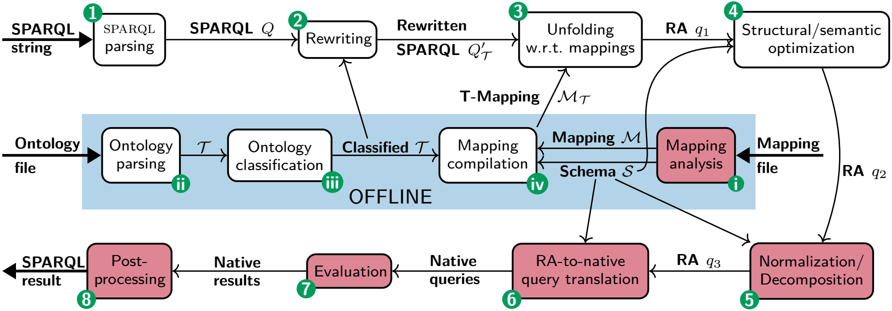
Retrieves all documents from the `bios` collection

G

(:/4 a :*Scientist*)
(:/4 :*firstName* "Kristen")
(:/4 :*lastName* "Nygaard")
(:/4 :*gotAward* :/4/Award/0)
(:/4 :*gotAward* :/4/Award/1)

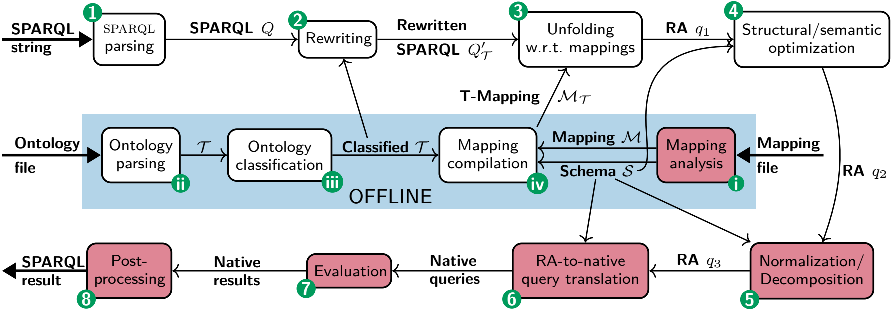
(:/4/Award/0 :*awardedInYear* 1999)
(:/4/Award/1 :*awardedInYear* 2001)
(:/4/Award/2 :*awardedInYear* 2001)
(:/4/Award/0 :*awardName* "Rosing Prize")
(:/4/Award/1 :*awardName* "Turing Award")

Updated architecture of Ontop



Steps (i) and (5)–(8) are specific to the underlying database system.

Updated architecture of Ontop



Steps (i) and (5)–(8) are specific to the underlying database system.

A prototype implementation for MongoDB:

- Mapping parser (i) and evaluation (7) are straightforward.
- Decomposition (5) extracts subqueries translatable into MUP(G)(L).
- Translation (6) is implemented according to [BCCRX16].
- Post-processing (8) converts the native result into SPARQL result.

unibz

References I