OBDA Beyond Relational DBs: A Study for MongoDB

Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk, and Guohui Xiao

Free University of Bozen-Bolzano, Italy, *lastname*@inf.unibz.it

Abstract. The database landscape has been significantly diversified during the last decade, resulting in the emergence of a variety of non-relational (also called NoSQL) databases, e.g., XML and JSON-document databases, key-value stores, and graph databases. To facilitate access to such databases and to enable data integration of non-relational data sources, we generalize the well-known ontology-based data access (OBDA) framework so as to allow for querying arbitrary databases through a mediating ontology. We instantiate this framework to MongoDB, a popular JSON-document database, and implement an prototype extension of the virtual OBDA system *Ontop* for answering SPARQL queries over MongoDB.

1 Introduction

Accessing data using native query languages is getting a more and more involved task for users, as databases (DBs) increase in complexity and heterogeneity. The Ontology-Based Data Access (OBDA) paradigm [10] has emerged as a proposal to simplify this kind of access, by allowing users to write high-level ontological queries, which in the classical virtual approach are translated automatically into low-level queries that DB engines can handle. This separation of concerns between the conceptual level and the DB level has been proven successful in practice, notably when data sources have a complex structure and end-users have domain but not necessarily data management expertise [5,4,1]. The OBDA approach is implemented by connecting a DB to an ontology by means of mappings, where traditionally the ontology is expressed in the OWL 2 QL profile of the Web Ontology Language OWL 2 [7], the queries are formulated in SPARQL, the Semantic Web query language, and the DB is assumed to be relational [3].

As envisioned by Stonebraker and Cetintemel [13], a multitude of DB architectures is needed to satisfy the needs of a wide variety of modern applications. This has been confirmed by the significant diversification of the DB landscape during the last decade. Some of these architectural changes have been proposed within the scope of relational DBs (e.g., column-oriented storage), while many have gone beyond them, causing the emergence of the so-called *NoSQL* (not only SQL) DBs. These *non-relational* DBs usually adopt one of four main data models, namely the column-family, key-value, document, and graph data models, and provide an (intimidating) number of query languages with varying querying capabilities. Notably, many of these new languages are limited [9] and thus clients might need to set up compensating post-processing techniques so as to satisfy advanced information needs.

We illustrate this novelty with a popular and representative instance of document DBs, MongoDB (`https://docs.mongodb.org/manual/`), which has rich but not yet full-fledged querying capabilities.

```
{ "_id": 4,
  "awards": [
    {"award": "Rosing Prize", "year": 1999, "by": "Norwegian Data Association"},
    {"award": "Turing Award", "year": 2001, "by": "ACM" },
    {"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"}
}
```

Fig. 1. A sample MongoDB document in the `bios` collection.

Example 1. MongoDB stores data in collections of semi-structured JSON[1]-style documents. A sample MongoDB document consisting of (possibly nested) key-value pairs and arrays is given in Figure 1. Thanks to its tree structure, this document gathers all relevant information about Kristen Nygaard, thus providing an excellent level of locality. In fact, it contains not only classical personal information (name, birth, etc.) but also information about the received awards. Note that in a normalized relational DB, this data would be spread across multiple tables. Instead, grouping closely related information adequately can significantly reduce the number of joins needed for answering queries.

MongoDB provides rather rich querying capabilities by means of the *aggregation framework*. In fact, consider a collection `bios` of documents as in Figure 1 storing information about prominent computer scientists, their names, dates of birth(/death), their contributions, and received awards. Then we can retrieve all persons who received two awards in the same year by the following aggregation framework query:

```
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" }}
])
```

We observe that this query performs a join within one document (in multiple stages). ■

To let users query non-relational data sources at the conceptual level (with SPARQL), in this paper, we propose to extend the OBDA framework to non-relational DBs. The following example highlights differences between the (procedural) low-level MongoDB query in Example 1 and a corresponding (declarative) high-level SPARQL query.

Example 2. Assume an ontology *Scientist* describing the `bios` information by means of roles *gotAward*, *awardedInYear*, *awardName*, *lastName*, *firstName* with the straightforward meaning, and connected to MongoDB by appropriate mappings. Then the query in Example 1 can be expressed by the following SPARQL query.

[1] JSON, or JavaScript Object Notation, is a tree-shaped format for structuring data.

```
SELECT ?firstName ?lastName ?awardName1 ?awardName2 ?year
WHERE { ?scientist :firstName ?firstName . ?scientist :lastName ?lastName .
        ?scientist :gotAward ?aw1 .          ?scientist :gotAward ?aw2 .
        ?aw1 :awardedInYear ?year .          ?aw2 :awardedInYear ?year .
        ?aw1 :awardName ?awardName1 .        ?aw2 :awardName ?awardName2 .
        FILTER (?aw1 != ?aw2) }
```

Observe that SPARQL abstracts away the complexity of the underlying query language and how data is structured in the DB. Instead it focuses on describing the relationship between entities of interest using the vocabulary provided by the ontology. ■

The contributions of this paper can be summarized as follows. To provide a uniform and well-founded way to access arbitrary DBs, we introduce a generalized OBDA framework for a class $\mathbf{D}$ of DBs. This generalization relies on the notion of a *relational wrapper* for $\mathbf{D}$, which is a function that represents the results of native queries as relations, and thus provides a uniform view of non-relational queries. Towards the virtual OBDA approach in the generalized setting, we revise the virtual OBDA architecture, and adapt from the relational case the translation algorithm from SPARQL queries to SQL queries [6] that uses relational algebra (RA) as an intermediate representation of the queries. The adaptation for the non-relational case relies on two translations: *(i)* SPARQL-to-RA, which makes use of the relational wrapper, and *(ii)* RA-to-$\mathcal{Q}_{\mathbf{D}}$, where $\mathcal{Q}_{\mathbf{D}}$ is the native-query-language for $\mathbf{D}$, which needs to be defined for each $\mathbf{D}$. Then, we instantiate this framework to the case of MongoDB using the formalization in [2]. Next, we implement a prototype system for answering SPARQL 1.0 queries under OWL 2 QL entailment regime over MongoDB by extending the virtual OBDA system *Ontop* [3], and provide some details about the implementation, which employs the translation from RA to MongoDB queries in [2]. Finally, we review other approaches for accessing data over non-relational DBs.

2 Preliminaries

MongoDB by Examples. As mentioned, MongoDB stores collections of documents, see Figure 1. Roughly, a collection corresponds to a table in a relational DB, and a document corresponds to a tuple (a row in a table). Recall that a document is an object that consists of key-value pairs, where a value can be an *atomic value* (e.g., `"Kristen"`), a *nested object* (e.g., `{"first": "Kristen", "last": "Nygaard"}`), or an *array* of values (e.g., `["OOP", "Simula"]`). Notice that, in MongoDB, the term 'key' is used with the meaning of 'attribute' in relational DBs, hence it should not be confused with the traditional notion of 'key constraint'. Here we adopt the same terminology. Thus, a key-value pair can be seen as the column name and the corresponding record in a tuple. A *path* is a concatenation of keys with '.' used to separate component keys, e.g., `awards`, `awards.year`, `birth`, `name.first` in the document in Figure 1.

MongoDB has a powerful querying mechanism provided by the aggregation framework, in which a query consists of a pipeline of *stages*, each transforming a set of documents into a new set of documents. We call this transformation pipeline an *aggregate query*. (There is also a basic querying mechanism in the form of *find* queries that can be expressed by aggregate queries, so we do not consider them here.) In this paper we are interested in 5 stages: *(i)* the *match* stage filters out input documents according

to some (Boolean) *criteria*; *(ii)* the *project* stage can specify which key-value pairs (not necessarily present in the input documents) should be present in the output; *(iii)* the *unwind* stage allows us to 'flatten' arrays by introducing a new document for every element in the array; *(iv)* the *group* stage combines different documents into one; *(v)* the *lookup* stage joins current trees with trees from an external collection.

Example 3. The following aggregate query selects from the `bios` collection the documents talking about scientists whose first name is Kristen, and for each document only returns the full name (using new keys) and the date of birth.

```
db.bios.aggregate([
    {$match: {"name.first": {$eq: "Kristen"}} },
    {$project: {
         "birth": true, "firstName": "$name.first", "lastName": "$name.last" } }
])
```

When applied to the document in Figure 1, it returns the following (shallow) tree:

```
{"_id": 4, "birth": "1926-08-27", "firstName": "Kristen", "lastName": "Nygaard"}
```

By default the document identifier `_id` is included in the answer of the query. ■

Example 4. Consider the query in Example 1, which is an aggregate query consisting of 6 stages that retrieves from the collection `bios` all persons who received two awards in one year. The first stage (project) flattens the complex object `name`, creates two copies of the array `awards`, and projects away all other fields. The second and third stages (unwind) flatten the two copies (`award1` and `award2`) of the awards array, which intuitively creates a cross-product of the awards. The fourth step (project) compares awards pairwise and creates a new key (`twoInOneYear`) whose value is true if the scientist has two awards in the same year. The fifth one (match) selects the documents where `twoInOneYear` is true, and the final stage (project) renames and projects keys.

By applying the query to the document in Example 1, we obtain:

```
{   "_id": 4,
    "firstName": "Kristen",
    "lastName": "Nygaard",
    "awardName1": "IEEE John von Neumann Medal",
    "awardName2": "Turing Award",
    "year": 2001
}
```

Due to space limitations, we do not provide an example with group and lookup. ■

Ontology-based Data Access Paradigm. In traditional OBDA, one provides access to an (external) relational DB through an ontology, which is connected to the DB by means of mappings.

Given a source schema $\mathcal{S}$ and an ontology $\mathcal{T}$, a (GAV) *mapping assertion* between $\mathcal{S}$ and $\mathcal{T}$ is an expression of the form $q(\boldsymbol{x}) \rightsquigarrow (f(\boldsymbol{x})\ \texttt{rdf:type}\ A)$ or $q'(\boldsymbol{x}, \boldsymbol{x'}) \rightsquigarrow (f(\boldsymbol{x})\ P\ f'(\boldsymbol{x'}))$, where A is a class name, P is a property name, $q(\boldsymbol{x})$, $q'(\boldsymbol{x}, \boldsymbol{x'})$ are arbitrary (SQL) queries expressed over $\mathcal{S}$, and f, f' are template functions. An *OBDA specification* is a triple $\mathcal{P} = \langle \mathcal{T}, \mathcal{M}, \mathcal{S} \rangle$, where $\mathcal{T}$ is an ontology, $\mathcal{S}$ is a relational DB schema, and $\mathcal{M}$ is a mapping (a finite set of mapping assertions). $\mathcal{T}$ is typically a TBox formulated in the OWL 2 QL profile [7], which guarantees that queries formulated over the TBox can be rewritten into equivalent queries over the DB. An *OBDA instance* is

a pair $\langle \mathcal{P}, D \rangle$, where $\mathcal{P}$ is an OBDA specification and D is a DB instance satisfying $\mathcal{S}$. The semantics of $\langle \mathcal{P}, D \rangle$ is specified in terms of interpretations of the classes and properties in $\mathcal{T}$. We define it, by relying on the following RDF graph $\mathcal{M}(D)$ *generated by* $\mathcal{M}$ *from* D (which can also be viewed as an ABox):

$$\{(f(\boldsymbol{o})\ \texttt{rdf:type}\ A) \mid \boldsymbol{o} \in ans(\psi, D) \text{ and } \psi \rightsquigarrow (f(\boldsymbol{x})\ \texttt{rdf:type}\ A) \text{ in } \mathcal{M}\} \cup$$
$$\{(f(\boldsymbol{o})\ P\ f'(\boldsymbol{o'})) \mid (\boldsymbol{o}, \boldsymbol{o'}) \in ans(\psi, D) \text{ and } \psi \rightsquigarrow (f(\boldsymbol{x})\ P\ f'(\boldsymbol{x'})) \text{ in } \mathcal{M}\}.$$

Then, a *model* of $\langle \mathcal{P}, D \rangle$ is simply a model of the ontology $\langle \mathcal{T}, \mathcal{M}(D) \rangle$.

In OBDA, there are mainly two approaches to query answering. The first (*materialization*) approach is to materialize the RDF graph $\mathcal{M}(D)$, load it into a triplestore, and rely on the standard query answering engine provided by the triplestore. This approach involves an ETL (Extract, Transform, and Load) process, which can be expensive, especially when the underlying DBs are updated frequently. The second (*virtual*) approach avoids explicitly constructing the RDF graph. Instead, it is based on query rewriting techniques and relies on the underlying DB engine for query evaluation. Thus, when the user asks a SPARQL query to the OBDA system, the query is first rewritten to another SPARQL query taking into account the ontology, then translated into an (SQL) query over the DB using the mapping, evaluated by the DB engine, and finally the SQL result is converted into a SPARQL result (cf. Figure 2).

In practice, the translation is done in two steps (cf. Figure 4): *(i)* using the mapping, the SPARQL query is 'unfolded' into a relational algebra (RA) query by substituting each triple t with the union of all q such that $q \rightsquigarrow t'$ is in $\mathcal{M}$ and t' unifies with t (for simplicity, here we omit the template functions), then *(ii)* the obtained relational algebra query is optimized and translated into an actual SQL query. The optimization step heavily relies on the schema (e.g., primary and foreign keys, unique attributes) to perform a number of semantic optimizations (e.g., eliminating redundant joins) [3].

3 Generalized OBDA framework

In this section we propose a generalized virtual OBDA framework over arbitrary DBs. We instantiate this framework in Section 5, using MongoDB.

We consider fixed countably infinite sets $\mathbb{C}$ of DB values, and $\mathbb{E}_{\mathbf{D}}$ of elements built over $\mathbb{C}$, e.g., named tuples, trees, or XML documents. We assume to deal with a class $\mathbf{D}$ of DBs, where each DB in $\mathbf{D}$ is a finite subset of $\mathbb{E}_{\mathbf{D}}$, e.g., relational DBs, MongoDB, or XML DBs. Moreover, we assume that $\mathbf{D}$ comes equipped with:

- Suitable forms of constraints, which might express both information about the structure of the data in DBs of $\mathbf{D}$, e.g., the schema information in relational DBs, and "constraints" in the usual sense of DBs, e.g., primary and foreign key constraints for relational DBs. We call a collection of such constraints a $\mathbf{D}$-*schema*.
- A query language $\mathcal{Q}_{\mathbf{D}}$, such that, for each query $q \in \mathcal{Q}_{\mathbf{D}}$ and for each instance $D \in \mathbf{D}$, the answer $ans(q, D)$ of q over D is defined, and is itself a DB in $\mathbf{D}$.
- A relational *wrapper* $[\![\cdot]\!]_{\mathbf{D}}$, which is a function transforming a $\mathcal{Q}_{\mathbf{D}}$-query q into a new query $[\![q]\!]_{\mathbf{D}}$ that takes a DB in $\mathbf{D}$ and returns a relation over $\mathbb{C}$ (i.e., a relation

in first normal form)[2]. The role of the wrapper is to present $ans(q, D)$ as a relation, by actually computing a query that retrieves from D what can be considered as the relational representation of $ans(q, D)$. In particular, when q is the identity query, $[\![q]\!]_{\mathbf{D}}$ is the query computing the relational view of a DB $D \in \mathbf{D}$.

Having these building blocks at hand, we now define $\mathbf{D}$-mapping assertions and their semantics. We start by introducing the notion of variable-to-RDF-term map, which is a generalization of the RDF-term template in relational OBDA. We say that $f(x_1, \ldots, x_n)$ is an *RDF term constructor* if it is a (partial) function $f : \mathbb{C}^n \to \mathbf{I} \cup \mathbf{L}$, where $\mathbf{I}$ is the set of IRIs and $\mathbf{L}$ is the set of RDF literals. Then, a *variable-to-(RDF)-term map* κ (for variable $?X$) has the form $?X \mapsto f(x_1, \ldots, x_n)$. In the following, $\pi_{x_1,\ldots,x_n}$ denotes the standard projection operator of relational algebra.

Definition 1. *A* $\mathbf{D}$-mapping assertion m *is an expression* $q \rightsquigarrow_K h$ *where:*

- q *is a* $\mathcal{Q}_{\mathbf{D}}$*-query, called* source query*;*
- h *is an RDF triple pattern, called* target*, of the form* $(?X_1$ `rdf:type` $A)$ *or* $(?X_1\ P\ ?X_2)$*, where* A *is a class name, and* P *is a property name;*
- K *is a set of variable-to-term maps, one for each variable* $?X_i$ *appearing in* h*.*

The mapping assertion m *is* safe *if for each* $?X_i \mapsto f(x_1, \ldots, x_n)$ *in* K*, we have that the function* $\pi_{x_1,\ldots,x_n} \circ [\![q]\!]_{\mathbf{D}}$ *is well-defined.*

A $\mathbf{D}$-mapping $\mathcal{M}$ *is a finite set of* $\mathbf{D}$*-mapping assertions.*

Notice that, the mapping assertion m is safe if and only if, for each variable-to-term map $?X_i \mapsto f(x_1, \ldots, x_n)$ used by m, the wrapper $[\![\cdot]\!]_{\mathbf{D}}$, when applied to the source query q of m, returns a query producing a relation whose attributes contain $x_1, \ldots, x_n$.

Definition 2. *Let* $\mathbf{D}$ *be a class of DBs,* $m = q \rightsquigarrow_K h$ *a* $\mathbf{D}$*-mapping assertion, and* $D \in \mathbf{D}$*. The* RDF graph $m(D)$ generated by m from D *is defined as follows:*

- *when* $h = (?X_1$ `rdf:type` $A)$*, then*

$$m(D) = \{(s\ \texttt{rdf:type}\ A) \mid s = f(v_1, \ldots, v_n),\ \kappa = ?X_1 \mapsto f(x_1, \ldots, x_n),\ K = \{\kappa\},\ (v_1, \ldots, v_n) \in \pi_{x_1,\ldots,x_n}([\![q]\!]_{\mathbf{D}}(D))\ \}$$

- *when* $h = (?X_1\ P\ ?X_2)$*, then*

$$m(D) = \{(s\ P\ o) \mid s = f_1(v_1, \ldots, v_n),\ o = f_2(v'_1, \ldots, v'_m),\ \kappa_1 = ?X_1 \mapsto f_1(x_1, \ldots, x_n),\ \kappa_2 = ?X_2 \mapsto f_2(x'_1, \ldots, x'_m),\ K = \{\kappa_1, \kappa_2\},\ (v_1, \ldots, v_n) \in \pi_{x_1,\ldots,x_n}([\![q]\!]_{\mathbf{D}}(D)),\ (v'_1, \ldots, v'_m) \in \pi_{x'_1,\ldots,x'_m}([\![q]\!]_{\mathbf{D}}(D))\ \}$$

For a $\mathbf{D}$*-mapping* $\mathcal{M}$*, the RDF graph* $\mathcal{M}(D)$ *is defined as* $\bigcup_{m \in \mathcal{M}} m(D)$*.*

Now, as in the relational case, an *OBDA specification for* $\mathbf{D}$ is a triple $\langle \mathcal{T}, \mathcal{M}, \mathcal{S} \rangle$, where $\mathcal{T}$ is an ontology, $\mathcal{S}$ is a $\mathbf{D}$-schema, and $\mathcal{M}$ is a $\mathbf{D}$-mapping. An *OBDA instance for* $\mathbf{D}$ consists of an OBDA specification $\langle \mathcal{T}, \mathcal{M}, \mathcal{S} \rangle$ for $\mathbf{D}$ and an instance $D \in \mathbf{D}$ satisfying $\mathcal{S}$. The semantics of such an instance is derived naturally from the semantics of $\mathbf{D}$-mapping assertions.

[2] Discussing the specific form and properties of wrappers is outside the scope of this paper.

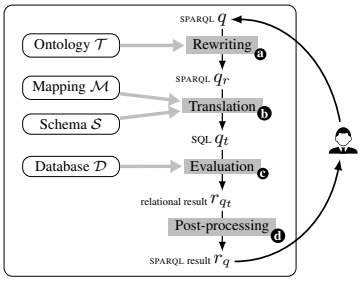

Fig. 2. Traditional virtual OBDA architecture

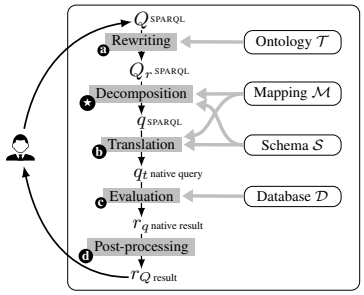

Fig. 3. Generalized virtual OBDA architecture

4 Generalized virtual OBDA architecture

The drawbacks of the materialization-based approach over the virtual approach to OBDA pointed out in Section 2 hold independently of the actual DB system used. Therefore, next we propose a generalized architecture of an OBDA system implementing the virtual approach to query answering, and we propose a two-step translation from SPARQL to the native query language.

Recall that in the virtual approach to relational OBDA, query answering consists of four steps: ⓐ rewriting, ⓑ translation, ⓒ evaluation, and ⓓ post-processing. In our generalized setting, the rewriting step ⓐ is exactly the same as in the relational case; the evaluation step ⓒ is simply delegated to the underlying DB engine. As for the translation step ⓑ, unlike in relational OBDA where an arbitrary SPARQL query can be translated into an SQL query, it cannot be guaranteed that every SPARQL query can be translated into a query in $\mathcal{Q}_{\mathbf{D}}$. For instance, many key-value stores do not natively support joins. Moreover, even though it might be possible to translate a query, the resulting query might be highly inefficient. Therefore, the following issues come up:

- Given a SPARQL query, can we translate it w.r.t. $\mathcal{M}$ and $\mathcal{S}$ into a $\mathcal{Q}_{\mathbf{D}}$-query?
- How can we decompose a SPARQL query into a set of $\mathcal{Q}_{\mathbf{D}}$-translatable subqueries so that the SPARQL query can be answered efficiently?

To address them, we modify the "traditional" virtual OBDA architecture depicted in Figure 2, obtaining the architecture depicted in Figure 3. The notable difference with respect to the relational architecture is an intermediate step between the rewriting step ⓐ and the translation step ⓑ, which we call *decomposition step* ★. This decomposition step gets as input the rewritten SPARQL query and decomposes it into a set of SPARQL subqueries, with the aim that each of these subqueries is translatable into a $\mathcal{Q}_{\mathbf{D}}$-query, which possibly is efficiently executable. We observe that, in order for a subquery to be translatable, the decomposition step might need to select only a subset of the mapping for its translation. The post-processing step ⓓ is now more involved than in the relational case, since it might also need to compose the results of intermediate subqueries according to SPARQL constructs (e.g., OPTIONAL, FILTER, JOIN).

Next, our goal is to adapt the two-step translation from the relational case to the generalized setting as in Figure 5. To this purpose, we show how to translate the input SPARQL query into an RA query using a $\mathbf{D}$-mapping (ⓘ in Figure 5) . Then, for an arbi-

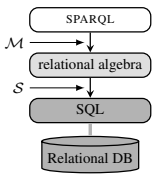

Fig. 4. Two-step SPARQL to native query translation

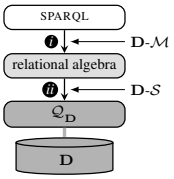

Fig. 5. Generalized two-step SPARQL to native query translation

trary $\mathbf{D}$, a translation from SPARQL to $\mathcal{Q}_{\mathbf{D}}$ can be obtained by providing a translation ⓘⓘ from RA to $\mathcal{Q}_{\mathbf{D}}$, which can be independent of the mapping component.

Let $\mathcal{M}$ be a $\mathbf{D}$-mapping. We now construct a relational mapping $\mathcal{M}_r$ and then define the translation of a SPARQL query q' to a RA query w.r.t. $\mathcal{M}$ to be the translation of q' w.r.t. $\mathcal{M}_r$ as defined in the relational case [10]. For a $\mathbf{D}$-query q, denote by R_q the name (and the corresponding signature) of the relational views $[\![q]\!]_{\mathbf{D}}(D)$, for $D \in \mathbf{D}$. The relational mapping $\mathcal{M}_r$ *induced by* the $\mathbf{D}$-mapping $\mathcal{M}$ and the wrapper $[\![\cdot]\!]_{\mathbf{D}}$ is defined as $\{R_q \rightsquigarrow K(h) \mid q \rightsquigarrow_K h \in \mathcal{M}\}$, where $K(h)$ is the triple resulting from substituting the variables in h with the corresponding RDF term constructor in K.

5 OBDA Framework over MongoDB

In this section we instantiate the generalized OBDA framework to MongoDB. This instantiation relies on work in [2], where we obtain the following results:

- We formalize a fragment of MongoDB aggregate queries that consists of match, unwind, project, group, and lookup stages, and that we call MUPGL. All example MongoDB queries in Sections 1 and 2 are MUPGL queries (actually, MUP queries, i.e., MUPGL queries without group and lookup stages).
- We propose a notion of MongoDB type constraints. These constraints allow one to specify that certain paths must point to an array (e.g., `awards`), or an atomic value (e.g., `name.last`), or an object (e.g., `name`).
- We define a relational view over MongoDB with respect to a set of type constraints.
- We develop a translation from RA expressions (over the relational view) to MUPGL queries. This translation shows that full RA can be captured by MUPGL, while RA over a single collection can be captured by MUPG.

We start by introducing a compact notation for MongoDB mapping assertions, which we also use to specify type constraints. We consider two extensions of paths called *array paths*. An *array index path* is a path where a key is followed by any combination of keys and '#', separated by dots. An *array element path* is the concatenation of an array index path with '.[#]'. Intuitively, for (simple) paths p_1 and p_2, any of the array paths p_1.#.p_2, p_1.#, or p_1.[#] imply that p_1 must point to an array. Moreover, the path p_1.#.p_2 (e.g., `awards.#.year`) is used to access the value of the path p_2 inside the array pointed to by p_1, while p_1.# (e.g., `awards.#`) is used to denote the indices and p_1.[#] (e.g., `contribs.[#]`) to denote the single elements of such an array. Hence, the presence of '#' or '[#]' requires that the path preceding it points to an array, whereas a

path that does not end with '#' or '[#]' must point to atomic values. We use *extended paths* to refer both to normal paths and to array paths.

The source queries we consider are a restricted form of MUP queries and can be represented as a pair (C, φ), where C is a collection name and φ is a criterion constructed using extended paths. Let K be a set of variable-to-term maps $?X \mapsto f(p_1, \ldots, p_n)$, where each p_i is an extended path. Then, a *MongoDB mapping assertion* is an expression of the form $(C, \varphi) \rightsquigarrow_K h$, where the variables in h constitute the domain of K. Here, we implicitly assume that two occurrences in K of '#' (or '[#]') preceded by the same extended path refer to the same array index.

Example 5. Consider the `bios` collection from Example 1. Let f_f=`{name.first}`, f_ℓ=`{name.last}`, f_x=:/`{_id}`, f_a=:/`{_id}`/Award/`{awards.#}`, f_y=`{awards.#.year}`, and f_n=`{awards.#.award}`. Denote the source query (`bios`, *true*) by q_s. Below is the MongoDB mapping $\mathcal{M}$ consisting of 6 mapping assertions from `bios` to the ontology *Scientist* that we assumed for the OBDA setting in Example 2.

1) $q_s \rightarrow_{\{?X \mapsto f_x\}}$ ($?X$ a :Scientist)
2) $q_s \rightarrow_{\{?X \mapsto f_x, ?F \mapsto f_f\}}$ ($?X$:firstName $?F$)
3) $q_s \rightarrow_{\{?X \mapsto f_x, ?L \mapsto f_\ell\}}$ ($?X$:lastName $?L$)
4) $q_s \rightarrow_{\{?X \mapsto f_x, ?A \mapsto f_a\}}$ ($?X$:gotAward $?A$)
5) $q_s \rightarrow_{\{?A \mapsto f_a, ?Y \mapsto f_y\}}$ ($?A$:awardedInYear $?Y$)
6) $q_s \rightarrow_{\{?A \mapsto f_a, ?N \mapsto f_n\}}$ ($?A$:awardName $?N$) ■

Given a MongoDB mapping $\mathcal{M}$, we first extract from it the MongoDB schema $\mathcal{S}_\mathcal{M}$ (a set of type constraints), and then define a relational wrapper $[\![\cdot]\!]_{\text{MongoDB}}$ for the source queries in $\mathcal{M}$. Let $A_\mathcal{M}$ be the set of all paths a such that there is a path of the form $p_1.\#.p_2$, $p_1.\#$, or $p_1.[\#]$ in $\mathcal{M}$, for an extended path p_1, and a is obtained from p_1 by removing all occurrences of '#' and '[#]', denoted $a = \mathit{sim}(p_1)$. Let $L_\mathcal{M}$ be the set of all paths ℓ such that there is a path of the form $p.k$ in $\mathcal{M}$, for an extended path p and a key k, and $\ell = \mathit{sim}(p.k)$. Let $O_\mathcal{M}$ be the set of all paths o such that there is a path of the form $p_1.k_1.k_2.p_2$ in $\mathcal{M}$, for extended (possibly empty) paths p_1, p_2 and keys k_1, k_2, and $o = \mathit{sim}(p_1.k_1)$. We say that $\mathcal{M}$ is *well-formed* if $A_\mathcal{M}$, $L_\mathcal{M}$, and $O_\mathcal{M}$ are mutually disjoint. The schema $\mathcal{S}_\mathcal{M}$ *implied* by a well-formed $\mathcal{M}$ is the set of type constraints stating that each path in $A_\mathcal{M}$ points to an array, each path in $L_\mathcal{M}$ points to an atomic (literal) value, and each path in $O_\mathcal{M}$ points to an object. Additionally, $\mathcal{S}_\mathcal{M}$ contains a type constraint stating that a path a in $A_\mathcal{M}$ points to an array of atomic values, if in $\mathcal{M}$ there is a path of the form $p.[\#]$ for $a = \mathit{sim}(p)$. Now, the relational wrapper $[\![\cdot]\!]_{\text{MongoDB}}$ for the source queries q in $\mathcal{M}$ is defined as the relational view with respect to $\mathcal{S}_\mathcal{M}$ of the result of evaluating q, cf. [2]. Intuitively, the signature of the relational views exported by the wrapper consists of all paths pointing to atomic values, and of the paths a.index, such that $a \in A_\mathcal{M}$ and in $\mathcal{M}$ there is a path of the form $p.\#$ for $a = \mathit{sim}(p)$.

Example 6. The signature of the relational view w.r.t. $\mathcal{S}_\mathcal{M}$, for $\mathcal{M}$ in Example 5 is $R_{\texttt{bios}}$(`_id`, `name.first`, `name.last`, `awards.index`, `awards.year`, `awards.award`).

The RDF graph $\mathcal{M}(\{d\})$, for the document d in Figure 1, is as follows:

(:/4 a :*Scientist*) (:/4/Award/0 :*awardedInYear* `1999`)
(:/4 :*firstName* `"Kristen"`) (:/4/Award/1 :*awardedInYear* `2001`)
(:/4 :*lastName* `"Nygaard"`) (:/4/Award/2 :*awardedInYear* `2001`)
(:/4 :*gotAward* :/4/Award/0) (:/4/Award/0 :*awardName* `"Rosing Prize"`)
(:/4 :*gotAward* :/4/Award/1) (:/4/Award/1 :*awardName* `"Turing Award"`)
(:/4 :*gotAward* :/4/Award/2) (:/4/Award/2 :*awardName* `"von Neumann Medal"`)

where :/4 is the URI of the object denoting Kristen Nygaard, and :/4/Award/0, :/4/Award/1, :/4/Award/2 are the URIs of the objects denoting his awards ■

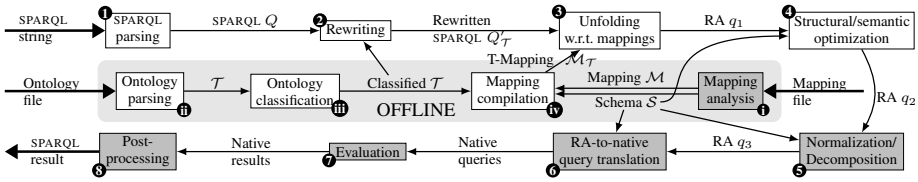

Fig. 6. Detailed SPARQL query answering process in *Ontop*

Since full RA can be captured by MUPGL [2], and SPARQL 1.0 can be translated into RA [6], it turns out that, given a mapping $\mathcal{M}$, every SPARQL 1.0 query *is* translatable w.r.t. $\mathcal{M}$ and $\mathcal{S}_{\mathcal{M}}$ into an MUPGL query. Note that this translation works in theory, but in practice, it might be still interesting to decompose input SPARQL queries. For instance, it might be more efficient to compute the union of two subqueries over different collections in the post-processing step than to delegate it to MongoDB.

6 Implementing generalized OBDA over MongoDB

We built a prototype implementation for answering SPARQL queries over MongoDB as an extension of the state-of-the-art OBDA system *Ontop* [3]. Below, we provide a description of the *Ontop* architecture and highlight which components require a new implementation in order to support MongoDB (or any other non-relational DB).

Architecture of *Ontop*. Query answering (QA) in *Ontop* under the OWL 2 QL entailment regime involves an offline and an online stage. The offline stage (highlighted in gray in Figure 6) takes as input the mapping and the ontology files and produces three entities used by the online QA stage: the classified ontology, the DB schema (extracted from the mapping file), and the so-called *T-mapping*, which is crucial for optimization and which is constructed by 'compiling' the classified ontology into the input mapping [11]. The online stage handles individual SPARQL queries, and can be split into 8 main steps: ① the input SPARQL query is parsed and ② rewritten according to the ontology, then it is ③ unfolded w.r.t. the T-mapping; ④ (the internal representation of) the resulting RA query is simplified by applying structural (e.g., replacing join of unions by union of joins) and semantic (e.g., redundant self-join elimination) optimization techniques; ⑤ the RA query is normalized and (possibly) decomposed so that it can be directly handled by the RA-to-native-query translator (e.g., renaming variables shared among multiple views in the relational case); ⑥ each normalized RA (sub)query is translated into a native query, which is then ⑦ evaluated by the DB engine; ⑧ the native results are post-processed into SPARQL results. These steps are related to the 5 steps of the generalized virtual OBDA architecture in Figure 3 as follows: rewriting ⓐ coincides with step ②. For practical reasons, decomposition ★ is implemented by step ⑤, which is part of translation ⓑ, implemented by steps ③–⑥. In fact, doing decomposition ★ before translation ⓑ would require to repeat some actions, e.g., unfolding w.r.t. mappings. Evaluation ⓒ and post-processing ⓓ correspond to steps ⑦ and ⑧, respectively.

Implementation for MongoDB. We observe that steps ⓘⓘ–ⓘⓥ and ①–④ are independent of the actual class $\mathbf{D}$ of DBs (white boxes in Figure 6), while steps ⓘ and ⑤–⑧ require specific implementations according to $\mathbf{D}$ (gray boxes). Therefore, our prototype

implements the latter five components. The mapping parser ⓘ and the RA-to-native-query translation ⑥ components are implemented according to what discussed in Section 5. The evaluation step ⑦ is straightforward. As for the decomposition step ⑤, it decomposes a given RA query q into subqueries q' in such a way that each q' can be translated into an MUP, or MUPG, or MUPGL query. Finally, the post-processing step ⑧ converts the result of a single MUPG query into SPARQL result (i.e., we are not yet able to compose according to the SPARQL constructs the results of multiple MUPG queries).

The current implementation supports MongoDB 3.2 and is able to return sound and complete answers to the subset of SPARQL queries that *(i)* correspond to BGPs with filters consisting of comparisons, and *(ii)* can be translated into MUPG queries. In particular, it can handle the SPARQL query in Example 2, whose translation w.r.t. the mapping in Example 5 is the MUP query in Example 1. We are now working on generating MUPGL queries for handling SPARQL 1.0 queries corresponding to BGPs with Optional, Filter (consisting of comparisons), and Order-by operators.

7 Related and future work

In [8], the authors study the problem of ontology-mediated query answering over key-value stores. They design a rule-based ontology language that uses keys as unary predicates, and where the scope of rules is at the record level (a record is a set of key-value pairs). Queries can be expressed as a combination of *get* and *check* operations, that roughly, given a path, return a set of values accessible via this path. Then they study the complexity of answering such queries under sets of rules. Strictly speaking, this work is still far from the OBDA setting because of the absence of mappings, and consequently, no distinction between user and native database query languages. Also note that their ontology and query languages are not compliant with any Semantic Web standard.

On the practical side, some proposals have been made to provide a uniform full-fledged query language for non-relational DBs. For instance, SQL^{++} [9] has been proposed as an extension of SQL with formal semantics to handle relations with arbitrary JSON-like nested values. It has been implemented in the virtual DB system FORWARD (`http://forward.ucsd.edu`), which supports MongoDB and other non-relational DBs, and allows for the creation of integrated views to hide heterogeneity between data sources. A closely related system is Apache Drill (`https://drill.apache.org`), which has a connector for MongoDB that currently uses the very limited MongoDB *find* query language and thus delegates most of the query answering to its post-processing components. These systems remain at the DB level so they could be embedded into an OBDA setting as the DB component. Note that the traditional relational OBDA framework is not sufficient to fully support these systems because they may have non first normal-form tables and views. In the future, we plan to design relational wrappers and RA-to-native query translators to support these systems in our generalized framework. We also plan to investigate the relationship between our framework and recent proposals for extending mappings towards additional datasources, such as RML (`http://rml.io/`), and for mapping JSON documents to RDF [12].

Acknowledgement. This work is partially supported by the EU under IP project Optique (*Scalable End-user Access to Big Data*), grant agreement n. FP7-318338.

References

1. N. Antonioli, F. Castanò, C. Civili, S. Coletta, S. Grossi, D. Lembo, M. Lenzerini, A. Poggi, D. Savo, and E. Virardi. Ontology-based data access: the experience at the Italian Department of Treasury. In *Proc. of CAiSE*, volume 1017 of *CEUR*, pages 9–16, 2013.
2. E. Botoeva, D. Calvanese, B. Cogrel, M. Rezk, and G. Xiao. A formal presentation of MongoDB (Extended version). CoRR Technical Report abs/1603.09291, arXiv.org e-Print archive, 2016. Available at `http://arxiv.org/abs/1603.09291`.
3. D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro, and G. Xiao. Ontop: Answering SPARQL queries over relational databases. *Semantic Web J.*, 2016. DOI: 10.3233/SW-160217.
4. D. Calvanese, P. Liuzzo, A. Mosca, J. Remesal, M. Rezk, and G. Rull. Ontology-based data integration in EPNet: Production and distribution of food during the Roman Empire. *Engineering Applications of Artificial Intelligence*, 2016.
5. M. Giese, A. Soylu, G. Vega-Gorgojo, A. Waaler, P. Haase, E. Jiménez-Ruiz, D. Lanti, M. Rezk, G. Xiao, Ö. L. Özçep, and R. Rosati. Optique: Zooming in on Big Data. *IEEE Computer*, 48(3):60–67, 2015.
6. R. Kontchakov, M. Rezk, M. Rodriguez-Muro, G. Xiao, and M. Zakharyaschev. Answering SPARQL queries over databases under OWL 2 QL entailment regime. In *Proc. of ISWC*, volume 8796 of *LNCS*, pages 552–567. Springer, 2014.
7. B. Motik, A. Fokoue, I. Horrocks, Z. Wu, C. Lutz, and B. Cuenca Grau. OWL Web Ontology Language profiles. W3C Recommendation, W3C, Oct. 2009. Available at `http://www.w3.org/TR/owl-profiles/`.
8. M.-L. Mugnier, M.-C. Rousset, and F. Ulliana. Ontology-mediated queries for NOSQL databases. In *Proc. of AAAI*, 2016.
9. K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ query language: Configurable, unifying and semi-structured. CoRR Technical Report abs/1405.3631, arXiv.org e-Print archive, 2014. Available at `http://arxiv.org/abs/1405.3631`.
10. A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *J. on Data Semantics*, X:133–173, 2008.
11. M. Rodriguez-Muro, R. Kontchakov, and M. Zakharyaschev. Ontology-based data access: Ontop of databases. In *Proc. of ISWC*, volume 8218 of *LNCS*, pages 558–573, 2013.
12. M. Sporny, G. Kellogg, and M. Lanthaler. JSON-LD 1.0. W3C Recommendation, W3C, Jan. 2014. Available at `https://www.w3.org/TR/json-ld/`.
13. M. Stonebraker and U. Cetintemel. "One size fits all": An idea whose time has come and gone. In *Proc. of ICDE*, pages 2–11, 2005.