Intelligenza Artificiale 1 (2019) 1–16
IOS Press

Ontology-based Data Access – Beyond Relational Sources

Elena Botoeva [a], Diego Calvanese [b,*], Benjamin Cogrel [b], Julien Corman [b], and Guohui Xiao [b]
[a] *Department of Computing, Imperial College London, 180 Queen's Gate London SW7 2AZ, U.K.*
E-mail: e.botoeva@imperial.ac.uk
[b] *Faculty of Computer Science, Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy*
E-mail: {calvanese,cogrel,corman,xiao}@inf.unibz.it

Abstract. The database (DB) landscape has been significantly diversified during the last decade, resulting in the emergence of a variety of non-relational (also called NoSQL) DBs, e.g., XML and JSON-document DBs, key-value stores, and graph DBs. To enable access to such data, we generalize the well-known ontology-based data access (OBDA) framework so as to allow for querying arbitrary data sources using SPARQL. We propose an architecture for a generalized OBDA system implementing the virtual approach. Then, to investigate feasibility of OBDA over non-relational DBs, we compare an implementation of an OBDA system over MongoDB, a popular JSON-document DB, with a triple store.

This article is an extended and revised version of an article that appeared in the proceedings of the *17th International Conference of the Italian Association for Artificial Intelligence* (AI*IA) [4].

Keywords: Ontology-based data access, NoSQL, JSON, MongoDB, query optimization

1. Introduction

To cope with the requirements of a variety of modern applications and their differing needs with respect to data management, in the last decade we have witnessed a strong diversification in the landscape of database (DB) management systems (DBMSs). Traditional relational DBMSs now coexist with so-called *NoSQL* ("not only" SQL) DBs, which redefine the format of the stored data, and how it is queried. These *non-relational* DBs usually adopt one of four main data models: *(i) Column-oriented DBMSs* maintain data tables similarly to traditional relational DBMSs, but store such tables by column, rather than by row. This allows for executing queries, which are expressed in traditional query languages like SQL, more efficiently for certain workloads. *(ii) graph databases* organize data in the form of elements (i.e., the nodes) connected by various relations (i.e., the edges), and are equipped with query languages based on graph navigation, such as SPARQL. *(iii) Key-value stores* represent data as a collection of key-value pairs, where keys are unique in a collection and are used to access the data. *(iv) Document stores* organize the data in documents, which have a hierarchical structure, are accessed via a key, and are encoded in some standard format, such as XML or JSON. Typically, they offer ad-hoc, in some cases quite expressive querying mechanisms (e.g., the aggregation framework of MongoDB), or even require writing JavaScript functions (e.g., CouchDB[1]). This wider choice of DBMSs offers the possibility to match application needs more closely, allowing for instance for more flexible data schemas, or more efficient (though simple) queries.

As a result, accessing data using native query languages is getting more and more involved for users. In this article, we rely on the *ontology-based data access* (OBDA) framework as a uniform solution to this

*Corresponding author. E-mail: calvanese@inf.unibz.it.

[1] http://couchdb.apache.org/

problem. The OBDA paradigm [20,26] has emerged as a proposal to simplify access to relational data for end-users, by letting them formulate high-level queries over a conceptual representation of the domain of interest, provided in terms of an ontology. In the classical *virtual* OBDA approach, data is *not* materialized at the conceptual level (which justifies the term "virtual"), and instead queries are translated automatically from the conceptual level into lower-level ones that DB engines can directly evaluate. The translation exploits a declarative specification of the relationship between the ontology and the data at the sources, provided in terms of *mapping assertions*. This separation of concerns between query formulation at the conceptual level and query execution at the DB level has proven successful in practice, notably when data sources have a complex structure, and end-users have domain knowledge, but not necessarily data management expertise [1,7,12]. Traditionally, in OBDA, the DB is assumed to be relational, the ontology is expressed in the OWL 2 QL profile of the Web Ontology Language OWL 2 [17], the mapping is specified in R2RML [9], and queries are formulated in SPARQL, the Semantic Web query language [13].

Extending the classical OBDA setting to arbitrary DBs requires to generalize some of its components. The first contribution of this work is to present such a generalized approach that enjoys all benefits already offered by OBDA. In particular, it allows for hiding from the user low-level concerns such as data storage and direct access to data (using the native query language of each data source), and it provides users with a high-level querying interface, closer to application needs. One could argue that OBDA is even more valuable in the NoSQL case compared to the relational one, as the gap between these low and high-level concerns tends to be wider. However, this extension also carries its own challenges, such as handling different data formats, the need for more advanced query optimization techniques due to lower-level query languages, or a possibly increased need for post-processing.

A second contribution is to investigate the applicability of the generalized OBDA framework in the practically significant case where the data source is a document store that offers rich querying capabilities, so that it is in principle feasible to fully delegate query answering to the source DB engine. In our investigation, we focus on MongoDB, a document-based DBMS, and one of the most popular NoSQL DBMSs as of today. MongoDB can be queried via a very expressive language, the so-called MongoDB *aggregation framework*, which has a more procedural flavor than SQL or SPARQL, and therefore can be complex to manipulate. Such a setting appears particularly well-suited for exploiting the added value offered by the OBDA paradigm.

Document-based DBMSs can also leverage the *denormalized* structure of their data: a document-based DB instance (i.e., a collection of documents) can often be seen as a denormalized version of a relational DB instance (where some joins are pre-computed). Therefore a natural question is whether OBDA over MongoDB can take advantage of such structure in order to answer queries efficiently, while at the same time offering a more user-friendly query language. As a third contribution of this work, we provide support for a positive answer. We do so by instantiating the generalized OBDA framework over MongoDB as an extension of the OBDA system *Ontop* [6], and comparing its performance with a triple store, which does not benefit from such denormalization. We adopt the triple store Virtuoso [11], using as dataset an instance of the well-known Berlin SPARQL Benchmark (BSBM) [3].

The rest of the article is structured as follows. In Section 2, we recall the standard OBDA framework over relational data sources. In Section 3, we introduce our proposal for generalizing OBDA to access arbitrary DBs, and present the architecture of a generalized OBDA system. In Section 4, we introduce MongoDB, describe our extension of *Ontop* over MongoDB, and illustrate the generalized OBDA architecture with a running example. In Section 5, we evaluate the performance of this system and compare it to the triple store Virtuoso using BSBM as dataset. In Section 6, we discuss related work, and we conclude the article with Section 7.

2. Ontology-based Data Access

We recall the traditional OBDA paradigm for accessing relational DBs through an ontology [26]. An *OBDA specification* is a triple $\mathcal{P} = \langle \mathcal{T}, \mathcal{M}, \mathcal{S} \rangle$, where $\mathcal{T}$ is an *ontology* modeling the domain of interest in terms of classes and properties, $\mathcal{S}$ is a relational DB schema, and $\mathcal{M}$ is a mapping consisting of a finite set of mapping assertions. We note that here we use the term "ontology" to denote a set of axioms involving only classes and properties, but not mentioning individuals. In other words, $\mathcal{T}$ consists only of the intensional part (typically called *TBox*) of an ontology in the sense of OWL 2. This choice is motivated by the

fact that in OBDA, the extensional component (typically called *ABox*) is provided by the DB instance via the mappings, as illustrated below.

To define mapping assertions, we make use of (RDF) *term constructors*, each of which is a function $f(x_1, \ldots, x_n)$ mapping a tuple of DB values to an IRI or to an RDF literal. Given a DB schema $\mathcal{S}$ and an ontology $\mathcal{T}$, a *mapping assertion* between $\mathcal{S}$ and $\mathcal{T}$ is an expression of one of the forms

$$\varphi(\boldsymbol{x}) \rightsquigarrow (f(\boldsymbol{x})\ \texttt{rdf:type}\ A), \quad \text{or}$$
$$\varphi(\boldsymbol{x}, \boldsymbol{x}') \rightsquigarrow (f(\boldsymbol{x})\ P\ f'(\boldsymbol{x}')),$$

where A is a class name in $\mathcal{T}$, P is a (data or object) property name in $\mathcal{T}$, $\varphi(\boldsymbol{x})$ and $\varphi(\boldsymbol{x}, \boldsymbol{x}')$ are arbitrary (SQL) queries expressed over $\mathcal{S}$, and f and f' are term constructors [15,20]. Mapping assertions allow one to define how classes and properties in $\mathcal{T}$ should be populated with values in a DB instance of $\mathcal{S}$ and with objects constructed from such values via the term constructors.

An *OBDA instance* is a pair $\langle \mathcal{P}, D \rangle$, where $\mathcal{P} = \langle \mathcal{T}, \mathcal{M}, \mathcal{S} \rangle$ is an OBDA specification and D is a DB instance satisfying $\mathcal{S}$. The semantics of $\langle \mathcal{P}, D \rangle$ is given with respect to the RDF graph $\mathcal{M}(D)$ *induced by* $\mathcal{M}$ *and* D, defined by

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

where $ans(\varphi, D)$ denotes the result of the evaluation of φ over D. Then, we define a *model* of $\langle \mathcal{P}, D \rangle$ to be simply a model of $\mathcal{T} \cup \mathcal{M}(D)$, i.e., a first-order interpretation that satisfies all axioms in $\mathcal{T}$ and all facts in $\mathcal{M}(D)$. We observe that $\mathcal{M}(D)$ provides a set of extensional facts, but such facts are typically kept *virtual*, i.e., they are not actually materialized.

Queries are usually formulated in SPARQL, the Semantic Web query language that allows for formulating expressive high-level queries over an RDF graph [13,19]. Such queries are answered over an OBDA instance $\langle \mathcal{P}, D \rangle$ according to the semantics of the chosen SPARQL *entailment regime*, considering $\mathcal{T}$ as the ontology, and $\mathcal{M}(D)$ as the RDF graph. Typically, in OBDA, the ontology $\mathcal{T}$ is expressed in OWL 2 QL, and the corresponding entailment regime is that of OWL 2 QL [14]. We denote with $ans_{\text{QL}}(q, \langle \mathcal{P}, D \rangle)$ the answers to a SPARQL query q over an OBDA instance $\langle \mathcal{P}, D \rangle$ according to the OWL 2 QL entailment regime.

3. Generalized OBDA Framework

In this section, we introduce a generalization of the OBDA framework to arbitrary DBs, and then propose an architecture for a generalized OBDA system.

3.1. OBDA over Arbitrary Databases

We assume to deal with a class $\mathbf{D}$ of DBs, e.g., relational DBs, XML DBs, or JSON stores, such as MongoDB. Moreover, we assume that $\mathbf{D}$ comes equipped with:

- Suitable forms of constraints, which might express both information about the structure of the stored data, e.g., the relational schema information in relational DBs, and "constraints" in the usual sense of relational DBs, e.g., primary and foreign keys. We call a collection of such constraints a $\mathbf{D}$-*schema* (or simply, *schema*).
- A way to provide a (flat) relational view to $\mathbf{D}$-schemas and $\mathbf{D}$-instances satisfying such schemas: for a $\mathbf{D}$-schema $\mathcal{S}$, $[\![\mathcal{S}]\!]$ is the corresponding relational schema, and for a $\mathbf{D}$-instance D satisfying $\mathcal{S}$, $[\![D]\!]$ is a relational DB over $[\![\mathcal{S}]\!]$. The function $[\![\cdot]\!]$ is called *relational wrapper*.
- A *native* query language $\mathcal{Q}$, such that, for a query $\varphi \in \mathcal{Q}$ and for a $\mathbf{D}$-instance D, the answer $ans(\varphi, D)$ to φ over D is defined (and is itself a $\mathbf{D}$-instance).

Now, given an ontology $\mathcal{T}$ and a $\mathbf{D}$-schema $\mathcal{S}$, a *mapping* $\mathcal{M}$ is a set of classical mapping assertions $\varphi \rightsquigarrow h$ between $[\![\mathcal{S}]\!]$ and $\mathcal{T}$, i.e., φ is a SQL query over $[\![\mathcal{S}]\!]$. Then, an *OBDA specification* is a triple $\langle \mathcal{T}, \mathcal{M}, \mathcal{S} \rangle$. This is analogous to the relational case, except that now $\mathcal{S}$ is a $\mathbf{D}$-schema (equipped with a relational wrapper) as opposed to a relational schema. An *OBDA instance* consists of an OBDA specification $\langle \mathcal{T}, \mathcal{M}, \mathcal{S} \rangle$ and a $\mathbf{D}$-instance D satisfying $\mathcal{S}$. The semantics of such an instance is derived naturally from the relational instance $[\![D]\!]$ corresponding to D via the relational wrapper $[\![\cdot]\!]$.

Note that our assumption that a relational wrapper is available for the class $\mathbf{D}$ of DBs is not restrictive in any way, since any form of data can be represented using relations, independently of how it is structured. Observe also that the source query in a mapping assertion in our generalized setting is not a native $\mathcal{Q}$ query, but a SQL query. Our framework has the advantage of having a uniform and expressive mapping language that is independent of $\mathbf{D}$ and $\mathcal{Q}$. It does not mean, however,

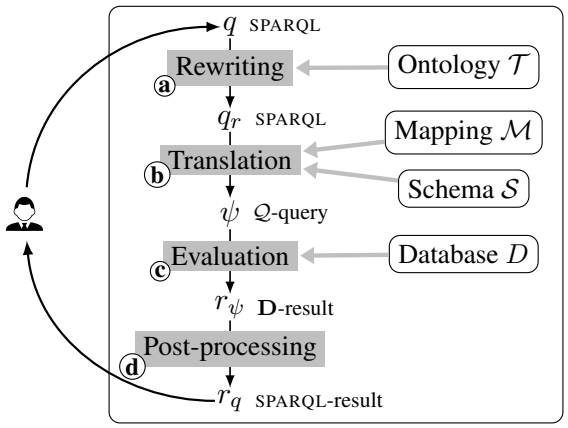

Fig. 1. Query Answering in OBDA

that the concrete user mapping language must strictly follow this specification. When it does not, the system should only be able to transform user mapping assertions into classical ones.

When referring to OBDA, we typically assume that it follows the *virtual* approach, in which materializing the RDF graph is avoided, and instead (part of) query answering is delegated to the DB. In this approach, the query answering process can be depicted as in Figure 1, and consists of 4 main steps: *(a)* An input SPARQL query q is first rewritten with respect to the ontology $\mathcal{T}$ into q_r (according to the semantics of the entailment regime, this step only rewrites the basic graph patterns (BGPs) in q [14]). *(b)* The rewritten SPARQL query q_r is translated into one or several native queries $\psi \in \mathcal{Q}$. When the DB engine does not support (efficiently) some SPARQL operators, multiple native queries might be required, and the evaluation of the unsupported operators may be postponed to the final post-processing step. *(c)* The native queries ψ are evaluated by the DB engine to produce $\mathbf{D}$-results r_ψ. *(d)* The results r_ψ of all queries ψ are combined and converted into the SPARQL result r_q in the post-processing step. In the generalized OBDA framework the post-processing step may be more involved than in the classical relational case, mostly due to the fact that the DB system may offer limited querying capabilities. In particular, some NoSQL DBs do not support joins. Another reason for not delegating certain query constructs to the DB is efficiency. For instance, in the case of nested data (e.g., JSON documents containing arrays), the unnesting (i.e., flattening) of nested objects into tuples may produce output objects that are much larger than the input, and so it may be preferable to perform unnesting as a post-processing step, so as to reduce network load between DB and client.

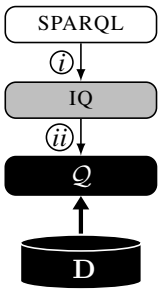

Fig. 2. SPARQL to native query translation

For the generalized OBDA framework, we propose to translate SPARQL queries to native queries in two steps (cf. Figure 2): first translate the input SPARQL query to an *intermediate query*, subject to transformations, and then translate the (transformed) intermediate query to a native query. The *intermediate query language*, denoted IQ, is expected to be a more high-level language than $\mathcal{Q}$, and can vary depending on $\mathcal{Q}$, but also on the considered fragment of SPARQL. On the one hand, it should at least capture such fragment (e.g., for BGPs, joins are sufficient, while for a fragment with property paths, IQ should include some form of recursion). On the other hand, IQ may include other operators that are present/expressible in $\mathcal{Q}$ (e.g., an unnest operator for dealing with nested data). Note that Relational Algebra (RA) as IQ is sufficient for the first-order fragment of SPARQL and for relational DBs. Our framework, relying on the use of IQ, provides several advantages: *(i)* it offers a better support for query optimization, since IQ, unlike SPARQL, can take into account the structure of the data, without necessarily being as low-level as $\mathcal{Q}$; *(ii)* the optimization techniques devised for IQ are independent of $\mathcal{Q}$; *(iii)* the translation from SPARQL to IQ is standard and depends only on the mapping (since IQ subsumes RA, such a translation has to extend the well-known translation from SPARQL to RA).

3.2. Architecture of an OBDA System over Heterogeneous Data Sources

We propose an architecture for an OBDA system able to answer SPARQL queries over heterogeneous data sources. This architecture, depicted in Figure 3, is composed of an *offline* stage, independent from the input SPARQL queries, and an *online* stage, dedicated to query answering.

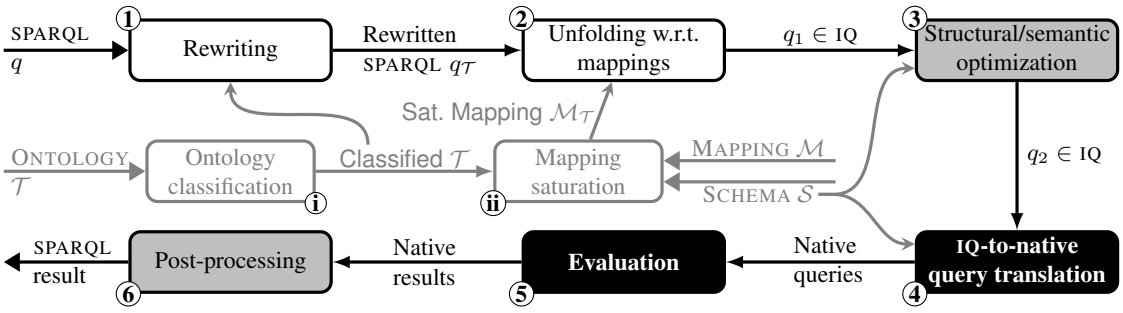

Fig. 3. Proposed architecture for an OBDA system

The offline stage consisting of steps ⓘ and ⓘⓘ, takes as input the ontology, mapping, and schema, and produces two elements, to be used during the online stage: the *classified* ontology, and the *saturated* mapping [21,23]. The former makes also implied inclusion assertions between classes and between properties explicit, while the latter is constructed by "saturating" the input mapping with the classified ontology. The saturation is obtained by adding to the existing mapping assertions additional ones that are derived by combining information from the input mapping and from the ontology axioms. For instance, if $\mathcal{M}$ contains an assertion $\varphi(\vec{x}) \rightsquigarrow (f(\boldsymbol{x})\ \texttt{rdf:type}\ A)$ and $\mathcal{T}$ an axiom $A\ \texttt{subClassOf}\ B$, then $\mathcal{M}_\mathcal{T}$ will contain also the assertion $\varphi(\vec{x}) \rightsquigarrow (f(\boldsymbol{x})\ \texttt{rdf:type}\ B)$. Saturating the mapping essentially allows us to consider ontology axioms already in the offline stage, and avoid (or reduce) their use during query rewriting. In this way, we anticipate to the offline stage operations that otherwise would need to be performed in the online stage, and this reduces the overall time required for query rewriting, when multiple user queries need to be executed over the ontology. We also observe that the saturated mapping can be significantly simplified for the online stage, by using query containment-based optimization to remove redundant mapping assertions.

The online stage handles individual SPARQL queries, and can be split into 6 main steps: ① the input SPARQL query is rewritten according to the classified ontology; ② the rewritten query is unfolded w.r.t. the saturated-mapping, by substituting each triple with its mapping definitions; ③ the resulting IQ 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; ④ the optimized IQ is translated into one or multiple native queries; ⑤ these are evaluated by the DB engine over the underlying DB (which is not explicitly shown as input in Figure 3); and finally, ⑥ the native results are combined and transformed into SPARQL results.

Such an architecture allows for steps ⓘ, ⓘⓘ, ①, and ② to be independent of the actual class $\mathbf{D}$ of DBs (white boxes in Figure 3). Steps ③ and ⑥ require an implementation specific to IQ (gray boxes), while ④ and ⑤ are specific to $\mathbf{D}$ (black boxes).

We emphasize that the structural and semantic optimization step is crucial for OBDA to work in practice. In particular, unfolded SPARQL queries often contain significantly more joins than actually necessary, since SPARQL atoms are triples, while data is typically stored in the form of n-ary entities (e.g., n-ary relations in relational DBs). In the case of OBDA over a document-based DB, these techniques can be extended to take advantage of additional opportunities for optimization offered by the structure of the DB instance. Some of these optimization techniques are illustrated on the example presented in Section 4.4.

4. OBDA over MongoDB

We illustrate the generalized OBDA framework by focusing on a specific NoSQL DB, namely MongoDB,[2] a popular and representative instance of document DBs. First, we introduce the data format and the query language of MongoDB, and we briefly relate them to the nested relational model and nested relational algebra. Then, we describe our prototype implementation for answering SPARQL queries over MongoDB. Finally, we illustrate the generalized OBDA framework over MongoDB on an example inspired by the BSBM benchmark.

[2] https://docs.mongodb.org/manual/

```
{ _id: 23226,
  name: "Olympus OM-D E-M10 Mark II",
  offers: [
    { offerId: 258, price: 747.14, year: 2015, vendor: {
        vendorId: 3785, name: "Yeppon Italia", homepage: "https://www.yeppon.it"}},
    { offerId: 895, price: 609.42, year: 2018, vendor: {
        vendorId: 481, name: "amazon.it", homepage: "https://www.amazon.it"}},
    { offerId: 922, price: 759.99, year: 2017, vendor: {
        vendorId: 481, name: "amazon.it", homepage: "https://www.amazon.it"}}
  ]
}

{ _id: 25887,
  name: "Panasonic Lumix DMC-GX80",
  offers: [
    { offerId: 311, price: 500.32, year: 2018, vendor: {
        vendorId: 481, name: "amazon.it", homepage: "https://www.amazon.it"}}
  ]
}
```

Fig. 4. A collection D^b of two MongoDB documents

4.1. MongoDB

MongoDB stores and exposes data as collections of JSON-like documents.[3] A sample collection of two MongoDB documents consisting of (nested) key-value pairs and arrays, is given in Figure 4, where each document contains information about a product: its id, name, and a list of offers, in the form of a JSON array. Each offer has itself an id, price, year, and vendor (in turn with id, name, and homepage).

In accordance with the generalized OBDA framework defined in Section 3, we assume that an input collection D of MongoDB documents complies to a schema $\mathcal{S}$. In other words, documents in D are expected to represent objects of the same type, and thus to follow the same structure.[4] So if a field (e.g., `offers` or `offers.vendor.homepage`) has an array (resp., an object or a constant) as value in one document, we assume that in every document this field either has an array (resp., an object or a constant) as value, or is absent (in which case the value is considered to be null).

Note that in a normalized relational DB instance, this data would be spread across several tables. Indeed, our example is inspired by the e-commerce scenario of the BSBM benchmark [3], where the data is structured according to a relational schema consisting of multiple tables. Figure 5 provides the relational view corresponding to this MongoDB collection, with distinct tables for products, vendors, and offers (the relational schema in the BSBM benchmark is actually more complex).

products

productId	name
23226	Olympus OM-D E-M10 Mark II
25887	Panasonic Lumix DMC-GX80

vendors

vendorId	name	homepage
481	amazon.it	http://www.amazon.it
3785	Yeppon Italia	http://www.yeppon.it

offers

offerId	price	year	product	vendor
258	747.14	2015	23226	3785
311	500.32	2018	25887	481
895	609.42	2017	23226	481
922	759.99	2018	23226	481

Fig. 5. Relational view $[\![D^b]\!]$ of the collection of Figure 4, following the BSBM schema

Note also that the JSON data in Figure 4 is *denormalized*. In particular, it contains redundant information: the name and homepage of vendor 481 are present 3 times. Document-based DBMSs like MongoDB can

[3] JSON (or *JavaScript Object Notation*) is a format for organizing data in tree-shaped structures. So in spirit it is similar to XML, but it is significantly more lightweight.

[4] This is not required by MongoDB itself, only by the OBDA framework.

take advantage of such redundancy. For instance, retrieving all vendors (with id, name, and homepage) of a given product over an instance of the relational schema of Figure 5 requires two (potentially costly) join operations. But the same request over the denormalized data does not require any join: the relevant information is already grouped within a document.

However, query execution can also be penalized by redundancy. For instance, a value for field `offers.vendor.vendorId` is always associated to the same value for field `offers.vendor.name`. But this type of constraint cannot be exploited by MongoDB (as of now) for query optimization. Therefore in order to retrieve the name(s) of vendor 481 for instance, MongoDB would fetch into memory all documents with an occurrence of 481 for field `offers.vendor.vendorId`, even though one document is sufficient in theory. Noticeably, this problem could be avoided by choosing a different document structure for the same data, with one document for each vendor rather than for each product, but with the drawback that the collection would then contain redundant information about products. In general, the choice of a particular document structure is a trade-off, favoring some queries, and penalizing others, and should be made according to the expected query workload (provided such information is available beforehand).

Like relational DBs, MongoDB allows for declaring indexes. By default, it creates a unique index over the (top-level) field `_id`, which serves as the primary key in a collection. Indexes can drastically speed up query execution. In particular, retrieving a (whole) document by a unique value of an indexed field (like the values of `offers.offerId` in Figure 4) can be done very efficiently by looking up the value in the index, and then fetching from disk data that is likely to be contiguous. On the other hand, queries on values with non-unique occurrences (e.g., the values of `offers.offer.vendorId`) may be less efficient, because multiple (non-contiguous) documents may need to be fetched.

MongoDB provides an ad-hoc querying mechanism for formulating expressive queries by means of the *aggregation framework*.[5] A *MongoDB aggregate query (*MAQ*)* is a sequence of *stages*, each of which takes one or two collections of documents as input, and produces another collection as output. This language is

[5] https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/

```
db.products.aggregate([
  {$project: {
     "name": true, "offer1": "$offers",
     "offer2": "$offers"}},
  {$unwind: "$offer1"},
  {$match: {"offer1.year": {$gte: 2016}}},
  {$unwind: "$offer2"},
  {$match: {"offer2.year": {$gte: 2016}}},
  {$project: {
     "name": true, "offer1": true,
     "offer2": true,
     "sameVendor": {$and: [
         {$ne: ["$offer1.offerId",
                "$offer2.offerId"]},
         {$eq: ["$offer1.vendorId",
                "$offer2.vendorId"]}]}},
  {$match: {"sameVendor": true}},
  {$project: {
     "product": {$concat:["bsbm:product/", _id]}
     "name": true,
     "vendorName": "$offer1.vendor.name",
     "price1": "$offer1.price",
     "price2": "$offer2.price"}}
])
```

Fig. 6. A MongoDB Aggregate Query (MAQ)

```
SELECT ?product ?productName
       ?price1 ?price2 ?vendorName
WHERE {
  ?product rdfs:label ?productName .
  ?offer1 bsbm:product ?product .
  ?offer2 bsbm:product ?product .
  ?offer1 bsbm:price ?price1 .
  ?offer2 bsbm:price ?price2 .
  ?offer1 bsbm:year ?year1 .
  ?offer2 bsbm:year ?year2 .
  ?offer1 bsbm:vendor ?vendor .
  ?offer2 bsbm:vendor ?vendor .
  ?vendor rdfs:label ?vendorName .
  FILTER (?offer1 != ?offer2 &&
          ?year1 >= 2016 && ?year2 >= 2016)
}
```

Fig. 7. A SPARQL query corresponding to the MAQ of Figure 6

powerful, but also more low-level (less declarative) than some well known query languages such as SQL or SPARQL. Because of this, MAQs can be complex to read and manipulate. As an illustration, the MAQ of Figure 6 retrieves all products offered twice by the same vendor since 2016. In comparison, the SPARQL query of Figure 7 retrieves the same information, but can be more easily understood. The more procedural

productsN

_id	name	offers					
		offerId	price	year	v.vendorId	v.name	v.homepage
23226	Olympus OM-D E-M10 Mark II	258	747.14	2015	3785	Yeppon Italia	https://www.yeppon.it
		895	609.42	2018	481	amazon.it	https://www.amazon.it
		922	759.99	2017	481	amazon.it	https://www.amazon.it
25887	Panasonic Lumix DMC-GX80	311	500.32	2018	481	amazon.it	https://www.amazon.it

Fig. 8. Nested relational view $[\![D^b]\!]_{\text{nested}}$ of the collection of Figure 4 (where `vendor` is abbreviated as `v`)

_id	name	offerId	price	year	v.vendorId	v.name	v.homepage
23226	Olympus OM-D E-M10 Mark II	258	747.14	2015	3785	Yeppon Italia	https://www.yeppon.it
23226	Olympus OM-D E-M10 Mark II	895	609.42	2018	481	amazon.it	https://www.amazon.it
23226	Olympus OM-D E-M10 Mark II	922	759.99	2017	481	amazon.it	https://www.amazon.it
25887	Panasonic Lumix DMC-GX80	311	500.32	2018	481	amazon.it	https://www.amazon.it

Fig. 9. Unnesting the sub-relation `offers` in the relation of Figure 8

v.vendorId	v.name	v.homepage	r				
			productId	name	offerId	price	year
3785	Yeppon Italia	https://www.yeppon.it	23226	Olympus OM-D E-M10 Mark II	258	747.14	2015
481	amazon.it	https://www.amazon.it	23226	Olympus OM-D E-M10 Mark II	895	609.42	2018
			23226	Olympus OM-D E-M10 Mark II	922	759.99	2017
			25887	Panasonic Lumix DMC-GX80	311	500.32	2018

Fig. 10. Nesting all attributes but `v.vendorId`, `v.name`, and `v.homepage` in the relation of Figure 9

flavor of MAQ also means that the sequence of stages of an MAQ is closer to its actual execution, whereas relational DBs/triple stores hide from the user the complexity of query planning (e.g., the ordering of joins). Hence, from a user perspective, OBDA over MongoDB appears indeed as a promising alternative to manually devising MAQs.

4.2. The Nested Relational Model

Alternatively, a collection of MongoDB documents can be viewed through the *nested relational model*, an extension of the relational model in which attributes can be also relation-valued, and not only atomic. Relation-valued attributes are called *sub-relations*. For instance, the MongoDB collection D^b of Figure 4 can be naturally represented in the nested relational model as the relation $[\![D^b]\!]_{\text{nested}}$ in Figure 8, with a sub-relation for the field `offers`.[6]

Nested relational algebra (NRA) [25] extends RA to operate on nested data. It is of particular interest for modeling operations on MongoDB collections, since it is equivalent in expressive power to a fragment of MAQ, as has been shown in [5]. NRA extends RA with two operators: *nest* and *unnest*. Intuitively, unnest flattens a sub-relation by concatenating each tuple in the sub-relation with the remaining attributes in the parent tuple. Instead, nest creates a sub-relation by partitioning the input relation, such that each element of the partition agrees on the values of the attributes that are not being nested. As an illustration, we first unnest the sub-relation `offers`, which yields the relation of Figure 9. Then we nest all attributes except for

[6]Notice though that the elements of a MongoDB array are ordered, whereas this is not the case of tuples in a sub-relation.

offer.vendor.vendorId, offer.vendor.vendorName, and offer.vendor.homepage into a sub-relation r, which yields the relation of Figure 10. As a result, tuples are grouped by vendor.

4.3. Instantiation of OBDA for MongoDB

We built a proof-of-concept prototype for answering SPARQL queries over MongoDB, called *Ontop/-MongoDB*, which extends the *Ontop* system [6] and implements the architecture described in Figure 3. The current implementation supports the fragment of SPARQL including BGPs, FILTER, JOIN, OPTIONAL, and UNION over MongoDB 3.4. In this implementation of the virtual OBDA architecture, NRA serves as IQ, and MAQ as the native query language. The system is designed to fully delegate query execution to the MongoDB engine,[7] thus minimizing the amount of post-processing required in step ⑥ of Figure 3.

Ontop/MongoDB takes as input (in addition to the MongoDB database instance) an OWL 2 QL ontology, a mapping, and a set of constraints. The constraints are user-defined *unicity constraints* (UCs) or *functional dependencies* (FDs) that hold over the JSON documents being queried. MongoDB may not be able to enforce such constraints, but they may nonetheless hold over the data. For instance, in the collection of Figure 4, an FD holds from offers.vendor.vendorId to offers.vendor.name, meaning that the value of the former determines the value of the latter. These constraints can be used for query optimization (e.g., to eliminate redundant joins, as illustrated in Section 4.4). We also emphasize that it can be verified whether a manually declared constraint actually holds over the data, by evaluating an appropriate query over the MongoDB instance. For instance, the MAQ of Figure 11 retrieves all sets of values (if any) that violate the FD from offers.vendor.vendorId to offers.vendor.name, in any MongoDB collection with the same schema as the collection of Figure 4.

Note also that if the MongoDB instance is a denormalized version of an existing relational DB instance, then UCs and FDs can be directly inferred from keys declared in the relational DB schema. For instance, let us assume that the MongoDB collection of Figure 4 is a denormalized version of the relational DB instance of Figure 5, and that the attribute vendorId is declared as the primary key of table

[7] An exception is the step that builds the returned RDF strings (IRIs and literals) from the constants retrieved from the DB.

```
db.products.aggregate([
  {$unwind: "$offers"},
  {$project: {
     "offers.vendor.vendorId": true,
     "offers.vendor.name": true}},
  {$group: {
     _id: "$offers.vendor.vendorId",
     names:
       {$addToSet: "$offers.vendor.name"}}},
  {$project: {count: {$size: "$names" }}},
  {$match: {count :{$gte: 2}}}
]);
```

Fig. 11. MAQ retrieving possible violations of the FD from offers.vendor.vendorId to offers.vendor.name, based on the schema of the collection of Figure 4

vendor. Then the FD from offers.vendor.vendorId to offers.vendor.name must hold over the MongoDB collection.

In step ③, in addition to applying relational optimization techniques implemented by *Ontop*, *Ontop/-MongoDB* also applies techniques specific to nested data, based on the equivalence with NRA mentioned above. In particular, it can take advantage of the UCs and FDs just mentioned.

In step ④, *Ontop/MongoDB* applies the NRA-to-MAQ translation given in [5]. An important consideration in this translation process is that one has to take into account the internal limitations that MongoDB puts on the size of in-memory intermediate results during query evaluation (currently 16 MB for a single document, and 100 MB for a collection). For example, a naive IQ-to-MAQ translation could produce an intermediate result in which the content of all input documents is merged into a single document, whose size might then exceed the memory limitations. Another key consideration is to take advantage of indexes available over the source JSON collection(s). Therefore *Ontop/MongoDB* does not apply the translation of [5] directly, but uses an optimized version, which makes the full delegation of query answering to MongoDB practically feasible.

4.4. Generalized OBDA by Example

We illustrate the generalized OBDA framework over MongoDB by elaborating on the running example inspired by the BSBM benchmark. The OBDA instance we consider is a pair $\langle \mathcal{P}^b, D^b \rangle$, where $\mathcal{P}^b = \langle \mathcal{T}^b, \mathcal{M}^b, \mathcal{S}^b \rangle$, the database instance D^b is the collection of documents given in Figure 4, and $\mathcal{S}^b$ is

```
SELECT * FROM products ⇝
  bsbm:product/{productId} rdfs:label {name} .

SELECT * FROM vendors ⇝
  bsbm:vendor/{vendorId} rdfs:label {name} ;
                         bsbm:homepage {homepage} .

SELECT * FROM offers ⇝
  bsbm:offer/{offerId} bsbm:price {price} ;
                       bsbm:year {year} ;
                       bsbm:product {product} ;
                       bsbm:vendor {vendor} .
```

Fig. 12. Mapping $\mathcal{M}^b$ over the relational view of Figure 5

the schema defining the structure of such documents. In addition, $\mathcal{S}^b$ contains two manually declared constraints, which hold over the data: *(i)* a UC for the field `offers.offerId`, meaning that each value of this field is unique in the whole collection;[8] *(ii)* an FD from the field `offers.vendor.vendorId` to the fields `offers.vendor.name` and `offers.vendor.homepage`.

We illustrate the evaluation over this OBDA instance of the SPARQL query q given in Figure 7. The example focuses on the steps that are most relevant for the generalization of the OBDA framework. In particular, we do not illustrate ontology classification, mapping saturation, and SPARQL query rewriting (respectively, steps ①, ⓘ, and ⓘⓘ in Figure 3), because these are identical to the case of OBDA over a relational database (for a detailed description, we refer to [14]). For this reason, we simplify the example by assuming that $\mathcal{T}^b$ consists only of a vocabulary (i.e., it contains no axiom), so that ontology classification, mapping saturation, and SPARQL query rewriting do not produce any change on the respective inputs.

Mapping. The mapping $\mathcal{M}^b$ is given in Figure 12. The SQL query φ in each mapping assertion is defined over the relational schema $[\![\mathcal{S}^b]\!]$ (corresponding to the DB instance $[\![D^b]\!]$ of Figure 5). For brevity, we use a set of RDF triples on the right-hand side of each mapping assertion. In such triples, $\{a\}$ is a placeholder for the value of attribute a in each tuple in $ans(\varphi, [\![D^b]\!])$, and $s_1\{a\}s_2$ stands for the concatenation of s_1, $\{a\}$, and s_2. In our case, since $\mathcal{T}^b$ is empty, the saturated mapping $\mathcal{M}^b_{\mathcal{T}^b}$ coincides with $\mathcal{M}^b$.

Note that the concrete mapping language currently used by *Ontop/MongoDB* supports source queries over $\mathcal{S}^b$ (rather than $[\![\mathcal{S}^b]\!]$), i.e., it defines JSON-to-RDF (rather than SQL-to-RDF) mappings. *Ontop/MongoDB* converts internally such mapping assertions into SQL-to-RDF ones. We only provide the latter here, to keep the exposition simple.

Unfolding into an IQ. The unfolding phase (step ② in Figure 3) starts with the SPARQL query $q_{\mathcal{T}^b}$, obtained by rewriting q w.r.t. the ontology $\mathcal{T}^b$ (since $\mathcal{T}^b$ is empty, $q_{\mathcal{T}^b}$ coincides with q in our case). Each triple pattern in $q_{\mathcal{T}^b}$ is substituted with the corresponding source SQL query in the saturated mapping $\mathcal{M}^b_{\mathcal{T}^b}$. This produces an RA query q_1, with the guarantee that for any DB instance D over the schema $\mathcal{S}^b$, $ans(q_1, [\![D]\!]) = ans(q_{\mathcal{T}^b}, \mathcal{M}^b_{\mathcal{T}^b}(D))$. This is in turn equivalent to $ans_{\text{QL}}(q, \langle \mathcal{P}^b, D\rangle)$, i.e., computing the answer to the original SPARQL query q over the OBDA instance according to the OWL 2 QL entailment regime can be reduced to evaluating q_1 over the relational view. The resulting query in our running example is given in Figure 13. As is conventional, symbols $\bowtie$, σ, and π respectively stand for inner join, selection, and (possibly complex) projection.

IQ *optimization.* As mentioned in Section 4.3, in our instantiation of the generalized OBDA framework, NRA serves as the IQ language.[9] In the IQ optimization phase (step ③ in Figure 3), *Ontop/MongoDB* rewrites the unfolded RA query, expressed over the relational schema $[\![\mathcal{S}^b]\!]$, into a semantically equivalent NRA query over the nested relational schema $[\![\mathcal{S}^b]\!]_{\text{nested}}$ (corresponding to the DB instance $[\![D^b]\!]_{\text{nested}}$ of Figure 8). While doing so, it applies a series of query optimization techniques, some of which are NRA-specific.

As an illustration, Figure 14 provides a naive rewriting over $[\![\mathcal{S}^b]\!]_{\text{nested}}$ of the query of Figure 13, where all sub-relations are simply unnested (in order to "access" their attributes), and no optimization is applied. Symbol χ stands for unnest, and $\chi_{\mathsf{of_i}\to(\_,\_,\_,\_,\_,\_)}$ for unnesting the sub-relation $\mathsf{of_i}$ (which counts 6 attributes). The resulting IQ requires a total of 9 unnest operations and 9 binary joins, whose execution may be costly.

In comparison, Figure 15 presents an optimized rewriting over $[\![\mathcal{S}^b]\!]_{\text{nested}}$, which contains no join, and only two unnest operations. A key technique to obtain this optimized rewriting is to identify redundant joins based on constraints holding over the schema $\mathcal{S}^b$. For instance, each value of `_id` can appear at most once in a MongoDB collection, therefore it determines the

[8] Recall that MongoDB also enforces an implicit primary key constraint on the field `_id`.

[9] Recall that NRA subsumes RA, hence an RA query is also an IQ.

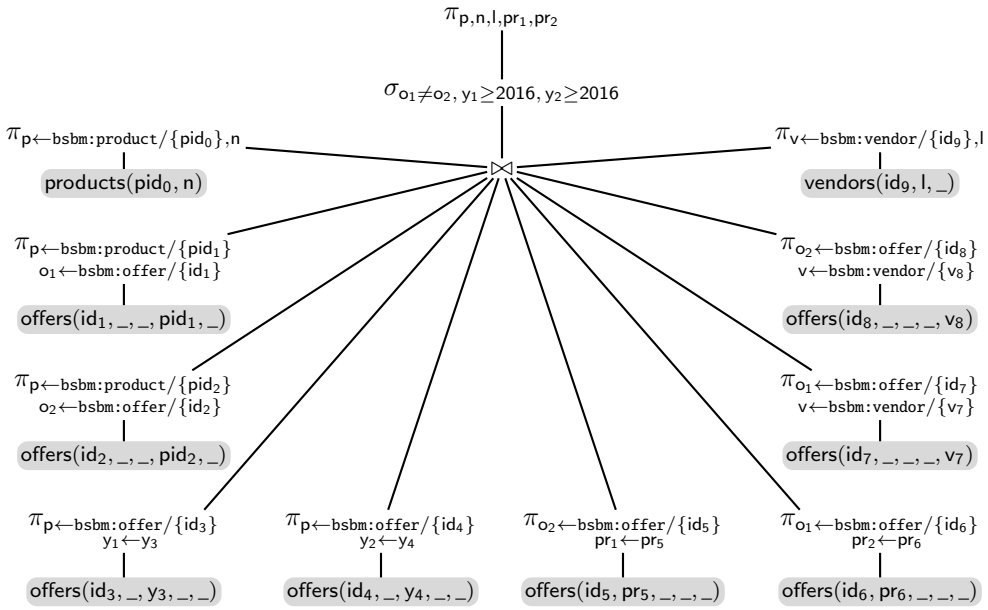

Fig. 13. Unfolding over $[\![\mathcal{S}^b]\!]$ of the query of Figure 7, w.r.t. the mapping of Figure 12

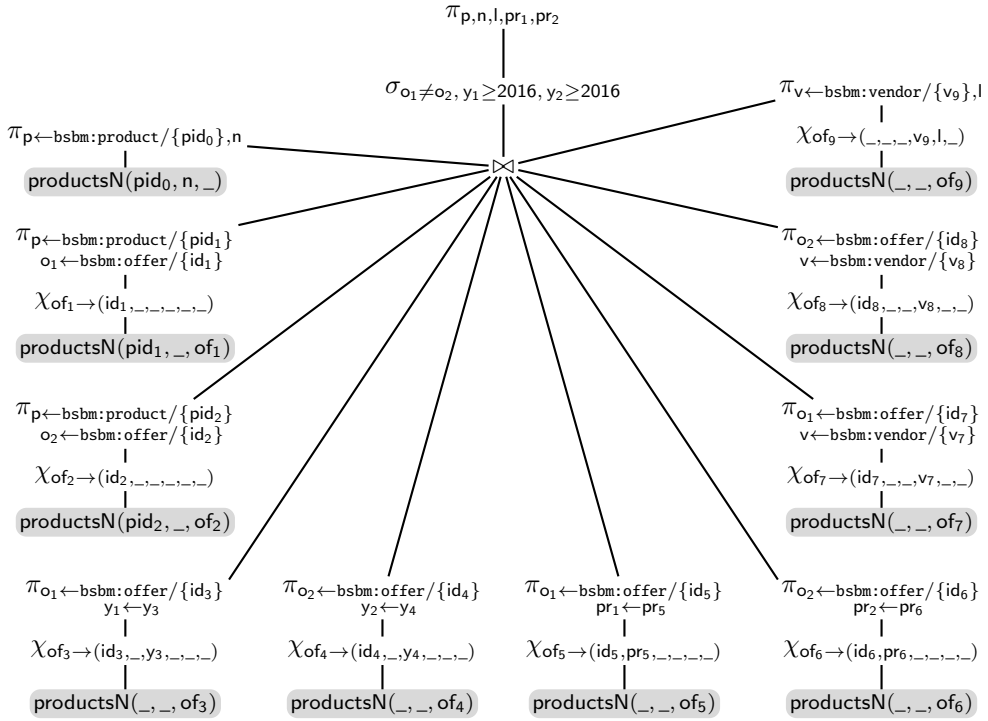

Fig. 14. Naive rewriting over $[\![\mathcal{S}^b]\!]_{\text{nested}}$ of the unfolded query of Figure 13

value of fields `name` and `offers`. This makes the join in Figure 14 between the first three operands unnecessary (all operands are numbered from 0 to 9, counterclockwise). Similarly, the UC declared for `offers.offerId` guarantees that each value for this field determines the value of the other 5 fields in the sub-relation. More interestingly, the declared FD guarantees that a value for `offers.vendor.vendorId` is always associated to

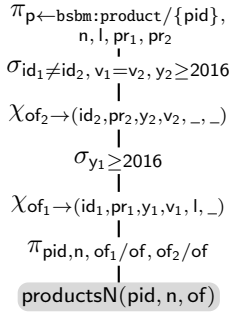

Fig. 15. Optimized rewriting over $[\![\mathcal{S}^b]\!]_{\text{nested}}$ of the unfolded query of Figure 13

the same value of `offers.vendor.name`, which makes the join between operands 7 and 9 (or 8 and 9) unnecessary. This holds even though a same value for `offers.vendor.vendorId` may appear multiple times in the collection. So in this last case, in order to eliminate the redundant join, *Ontop/MongoDB* exploits an FD which does not follow from a UC. This is an example of NRA-specific optimization, usually not needed for OBDA over relational DBs (assuming the database to be at least in third normal form).

IQ-*to-*$\mathcal{Q}$ *translation.* In the IQ-to-native query translation phase (step ④ in Figure 3), *Ontop/MongoDB* rewrites the optimized IQ q_2, expressed over $[\![\mathcal{S}^b]\!]_{\text{nested}}$, into an equivalent MAQ q_m over $\mathcal{S}^b$ (i.e., such that for every D satisfying $\mathcal{S}^b$, $ans(q_2, [\![D]\!]_{\text{nested}}) = ans(q_m, D)$), based on the correspondence between NRA and (a fragment of) MAQ established in [5].

Since the optimized IQ in our running example (Figure 15) does not contain binary operators, the conversion to MAQ is relatively straightforward: σ, π, and χ can be directly mapped to the MAQ operators `$match`, `$project`, and `$unwind` respectively, yielding the MAQ of Figure 6.

Once again, query optimization techniques are applied by *Ontop/MongoDB* in order to produce a more efficient MAQ. These are MAQ-specific, i.e., they could not be implemented at the level of IQ, and mostly pertain to the translation of binary NRA operators (including a dedicated planner).

We nonetheless illustrate one optimization technique that applies to our running example. MAQ allows for prefiltering a collection of MongoDB documents, based on the value of a nested field nf, without the need to unnest the corresponding sub-relation first. If nf is indexed, this pre-filter can significantly reduce the number of required disk accesses during query execution, fetching only documents in which some occurrence of nf satisfies the filter condition.

```
db.products.aggregate([
  {$match: {"offers.year": {$gte: 2016}}},
  {$project: {
     "name": true, "offer1": "$offers",
     "offer2": "$offers" }},
  {$unwind: "$offer1"},
  {$match: {"offer1.year": {$gte: 2016}}},
  {$unwind: "$offer2"},
  {$match: {"offer2.year": {$gte: 2016}}},
  {$project: {
     "name": true, "offer1": true,
     "offer2": true,
     "sameVendor": {$and: [
         {$ne: ["$offer1.offerId",
                "$offer2.offerId"]},
         {$eq: ["$offer1.vendorId",
                "$offer2.vendorId"]}]}},
  {$match: {"sameVendor": true}},
  {$project: {
     "product": {$concat:
         ["bsbm:product/",
          {$toString: "$_id"}]},
     "name": true,
     "vendorName": "$offer1.vendor.name",
     "price1": "$offer1.price",
     "price2": "$offer2.price"}}
])
```

Fig. 16. Optimized version of the MAQ of Figure 6

In our example, assuming that the field `offers.year` is indexed,[10] one can prepend a `$match` stage to the MAQ, in order to pre-filter products with at least one offer in the desired range of years ($\geq$ 2016), before unnesting the sub-relation. The resulting MAQ is given in Figure 16.

Native query evaluation and post-processing. The evaluation phase (step ⑤ in Figure 3) simply consists in evaluating the optimized MAQ over the MongoDB collection D^b. *Ontop/MongoDB* offers the possibilty to fully delegate query execution to the source engine, meaning that the results returned by the source DBMS can be immediately converted to SPARQL solution mappings.

Full delegation of query execution may not be possible for all DBMSs though, or it may not be desirable, as explained in Section 3.1. This is the reason for the post-processing phase (step ⑥ in Figure 16). As a rel-

[10] We also assume that this index is optimized for ordering, e.g., is a B-tree.

atively basic illustration, *Ontop/MongoDB* offers as an option to postprocess the construction of IRIs. In this case, the last stage of the MAQ in Figure 16 will not project the field `"product"`.[11]

5. Evaluation

We have carried out an evaluation that aims at determining whether OBDA over MongoDB is a realistic solution performance-wise, and in particular whether it is able to exploit the document structure of MongoDB collections. We focus on answering queries over datasets that do not fit into memory. In such a setting, a key concern for performance is to limit disk access, i.e., the number of non-contiguous pages that need to be fetched into memory.

To this end, we compare *Ontop/MongoDB* to the triple store Virtuoso [11], which represents a diametrically opposite approach to answering SPARQL queries, as far as the data and index structure are concerned. Indeed, Virtuoso stores data as quads (i.e., triples extended with the graph name), and for each element of the quads it maintains an extensive index structure, which is in particular highly optimized for retrieving (multiple) triples sharing a constant value[12]. In comparison, retrieving all documents for a given value of an indexed field may be inefficient in MongoDB if the value is not unique in the index, as it requires fetching multiple (non-contiguous) documents from disk. Instead, when the value is unique, MongoDB can fetch the whole document containing this value very efficiently, whereas for Virtuoso fetching the same data may require multiple disk accesses.

We expect the evaluation to reflect these differences, i.e: *(i)* that *Ontop/MongoDB* outperforms Virtuoso for queries containing a unique constant in an indexed field, and fetching a single document; *(ii)* that Virtuoso outperforms *Ontop/MongoDB* for queries whose constants have multiple occurrences in the JSON collection.

An additional goal of the experiments is to determine whether the cost of query rewriting itself (i.e., generating the MAQ) introduces an excessive overhead.

5.1. Dataset and Evaluation Environment

As dataset we used an instance of the well-known BSBM benchmark [3], which emulates an e-commerce scenario, centered on offered products. The number of products in the instance is 4 million, giving 1.2 billion RDF triples, whose total size is 156 GB.

BSBM also provides a representation of this dataset as a relational DB instance, composed of 10 tables (product, offer, vendor, etc.). Based on the relational schema of this instance, we generated a 118 GB collection of JSON documents containing the same data. The structure of the documents in this collection extends the one of Figure 4, grouping in each document all information pertaining to a single product.

The latest version of BSBM comes with 11 queries, numbered from 1 to 12 (there is no query 6 anymore). Among these, 3 were discarded, because they contain SPARQL features not (yet) supported by *Ontop/MongoDB* (DESCRIBE queries, *bound* operator, and variables over predicates). We instantiated 10 versions of each of the 8 remaining queries, replacing constant placeholders with values randomly sampled from the data. One version of each query was set aside for a cold run, and the $8 \cdot 9$ other instantiated queries were shuffled as a query mix. Execution times reported below are averaged over these 9 versions.

The systems being compared are Virtuoso v7.2.4 (over the RDF triples), and *Ontop/MongoDB* with MongoDB v3.4.2 (over the JSON collection). Queries were executed on a 24 cores Intel Xeon CPU at 3.47 GHz, with a 5.4 TB 15k RPM RAID-5 hard-drive cluster. 8 GB of RAM were dedicated to each system (MongoDB and Virtuoso) for caching and intermediate operations. The OS page cache was also flushed every 5 seconds, to ensure that each system could only exploit these 8 GB for caching. The query timeout was set to 500 s. For each constant appearing in a query, the corresponding field in the MongoDB collection was indexed.

An executable for *Ontop/MongoDB* is available online, together with the SPARQL queries, mapping, constraints, and both datasets (JSON and RDF), so that the experiment can be reproduced. The generated MAQs are also provided.[13]

[11] Note that the `_id` field is systematically returned by MongoDB, therefore it does not need to be explicitly retained by the projection.

[12] `http://docs.openlinksw.com/virtuoso/rdfperfrdfscheme/`

[13] `https://www.dropbox.com/sh/nz8dfas5ijpr76y/AACJzxHZUInrHi6Vq3Lk8f8ra?dl=0`

5.2. Results and Analysis

As a first element of answer, we observed that all MAQs generated by *Ontop/MongoDB* are optimal with respect to the document structure, in the sense that no unnecessary join is performed that could be in theory avoided.

Table 1 reports the execution times for both systems. For *Ontop/MongoDB*, we distinguish query rewriting time ("rw"), i.e., the time spent generating the MAQ, from its actual evaluation ("eval") by MongoDB. Rewriting time does not depend on the size of the data, but only on the query, mapping, ontology, and constraints, which are less likely to grow out of proportion. Still, for some of the cheaper MAQs (< 100 ms), this overhead represents the major part of the execution time. This can be partly explained by the wide range of optimizations performed by *Ontop/MongoDB*. But it is also an aspect to improve, for OBDA over MongoDB to be considered a viable alternative to MongoDB itsef, at least in applications with high performance requirements.

We now focus on query evaluation times. For each of the 9 versions of Query 5, the evaluation either timed out, or exceeded MongoDB's memory limitations (see Section 4.3). This is explained by the fact that this query contains an anti-join, which requires a (close to) full collection scan from MongoDB. For the 7 remaining queries, we observe a sharp contrast in performance between the two systems, which matches the above expectations. Queries 1 and 4 present a very favorable setting for Virtuoso: the SPARQL BGPs are of limited size (≤ 5 triple patterns), and each of them contains 3 constants. On the other hand, because none of these constants is unique in the JSON collection, the evaluation by *Ontop/MongoDB* requires fetching multiple documents from disk. As expected, for these two queries, evaluating the SPARQL query with Virtuoso was one order of magnitude faster than evaluating the corresponding MAQ with *Ontop/MongoDB*. As for the 5 remaining queries, they all represent a setting where MongoDB can fully benefit from denormalization. First, all 5 queries require data contained in one document only. In addition, they all contain a constant in an indexed field, where the index is either declared as a unique (Queries 2, 7, 8, and 10), or contain only unique values (Query 12). For each of these queries, the evaluation was one to two orders of magnitude faster for MongoDB. This confirms that *Ontop/MongoDB* was able to generate MAQs that take full advantage of the document (and index) structure.

6. Related Work

The idea of using wrappers to access external data sources dates back to the 90s; see e.g., the Garlic data integration system [22]. In recent years, several extensions of the SQL language have been proposed for accessing JSON data, e.g., SQL++ [18]. These query languages are either directly supported by the source DB engines (e.g., Couchbase[14] and AsterixDB[15]) or by middleware. In the particular case of a MongoDB data source, such middleware includes Drill[16], Dremio[17], Studio 3T[18], and the MongoDB Connector for Spark[19]. With such systems, users can query MongoDB collections as nested tables. SQL queries are automatically translated to (basic) MongoDB queries, and post-processing is often required to compute advanced query constructs. These middleware systems differ fundamentally from OBDA systems in the type of virtual data representation they expose: a representation whose structure is derived directly from the one of the data source in the case of middleware systems, and an indirectly mapped RDF graph in the case of OBDA.

Several mapping language proposals already exist that extend R2RML for converting non-relational data sources to RDF, e.g., RML [10], xR2RML [16], KR2RML [24], and D2RML [8]. These languages extend the relational model used in R2RML to more general cases (e.g., CSV, JSON, and Web Services). The corresponding systems are mostly used for data conversion; for instance, the implementation Morph-xR2RML also supports SPARQL query answering by partially materializing the relevant RDF graph.

Finally, the approach of [2] is comparable in spirit to ours, in that it also aims at delegating query execution to a NoSQL source engine, and relies on an object-oriented (OO) intermediate representation, similar to our "relational view". A key difference though is that the mapping is from the ontology vocabulary to the OO layer, rather than from the source DB to the ontology vocabulary. The aim is to simplify the mapping specification, and make it independent of the underlying source DB. The expressivity of such a mapping is thus limited, essentially mapping OWL classes to (possibly nested) relations.

[14] http://couchbase.com/
[15] https://asterixdb.apache.org/
[16] https://drill.apache.org/
[17] https://www.dremio.com/
[18] https://studio3t.com/knowledge-base/articles/sql-query/
[19] https://docs.mongodb.com/spark-connector/

Table 1
Execution times (ms) for *Ontop/MongoDB* and Virtuoso, over the BSBM benchmark (4 million products). Values are averaged over 9 versions of each query

	Query	1	2	4	5	7	8	10	12
Ontop/MongoDB	rw	26	179	102	NA	417	838	22	35
	eval	2672	43	3713	NA	53	66	34	40
Virtuoso	eval	258	308	403	1179	3995	1897	3966	327

7. Conclusions

In this article, we have presented a generalized OBDA framework for arbitrary (not only relational) DBs. It provides a convenient uniform querying interface, by means of a high-level vocabulary coupled with a familiar query language (SPARQL), as an alternative to the variety of ad-hoc native query languages of NoSQL DBMSs. We also propose a practical architecture for a generalized virtual OBDA approach, that allows for answering SPARQL queries over arbitrary data sources.

We have instantiated this framework in the specific case of MongoDB, as an extension (called *Ontop/MongoDB*) of the OBDA system *Ontop*, and we have compared its performance to that of a triple store. The evaluation we have carried out shows that *Ontop/MongoDB* is able to generate queries in MAQ, the native query language of MongoDB, that take full advantage of the denormalized structure of the data.

As a continuation of this work, we plan to evaluate the impact of the different techniques implemented within *Ontop/MongoDB* to optimize the generated MAQs, using a wider range of queries, but also different document structures for the same dataset.

References

[1] Natalia Antonioli, Francesco Castanò, Cristina Civili, Spartaco Coletta, Stefano Grossi, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Domenico Fabio Savo, and Emanuela Virardi. Ontology-based data access: the experience at the Italian Department of Treasury. In *Proc. of the Industrial Track of the 25th Int. Conf. on Advanced Information Systems Engineering (CAiSE)*, volume 1017 of *CEUR Workshop Proceedings,* `http://ceur-ws.org/`, pages 9–16, 2013.

[2] Thiago Henrique Dias Araujo, Barbara Tieko Agena, Kelly Rosa Braghetto, and Renata Wassermann. OntoMongo – Ontology-Based data access for NoSQL. In Mara Abel, Sandro Rama Fiorini, and Christiano Pessanha, editors, *Proc. of the 10th Seminar on Ontology Research in Brazil (OntoBras) and 1st Doctoral and Masters Consortium on Ontologies*, volume 1908 of *CEUR Workshop Proceedings,* `http://ceur-ws.org/`, pages 55–66, 2017.

[3] Christian Bizer and Andreas Schultz. The Berlin SPARQL benchmark. *Int. J. on Semantic Web and Information Systems*, 5(2):1–24, 2009.

[4] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Julien Corman, and Guohui Xiao. A generalized framework for ontology-based data access. In *Proc. of the 17th Int. Conference of the Italian Assoc. for Artificial Intelligence (AI*IA)*, volume 11298 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2018.

[5] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, and Guohui Xiao. Expressivity and complexity of MongoDB queries. In *Proc. of the 21st Int. Conf. on Database Theory (ICDT)*, volume 98 of *Leibniz Int. Proc. in Informatics (LIPIcs)*, pages 9:1–9:22, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[6] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering SPARQL queries over relational databases. *Semantic Web J.*, 8(3):471–487, 2017.

[7] Diego Calvanese, Pietro Liuzzo, Alessandro Mosca, Jose Remesal, Martin Rezk, and Guillem Rull. Ontology-based data integration in EPNet: Production and distribution of food during the Roman Empire. *Engineering Applications of Artificial Intelligence*, 51:212–229, 2016.

[8] Alexandros Chortaras and Giorgos Stamou. D2RML: Integrating heterogeneous data and web services into custom RDF graphs. In *Proc. of the Workshop on Linked Data on the Web (LDOW)*, volume 2073 of *CEUR Workshop Proceedings,* `http://ceur-ws.org/`, 2018.

[9] Souripriya Das, Seema Sundara, and Richard Cyganiak. R2RML: RDB to RDF mapping language. W3C Recommendation, World Wide Web Consortium, September 2012. Available at `http://www.w3.org/TR/r2rml/`.

[10] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. RML: A generic language for integrated RDF mappings of heterogeneous data. In *Proc. of the Workshop on Linked Data on the Web (LDOW)*, volume 1184 of *CEUR Workshop Proceedings,* `http://ceur-ws.org/`, 2014.

[11] Orri Erling and Ivan Mikhailov. RDF support in the Virtuoso DBMS. In *Networked Knowledge – Networked Media*, volume 221 of *Studies in Computational Intelligence*, pages 7–24. Springer, 2009.

[12] Martin Giese, Ahmet Soylu, Guillermo Vega-Gorgojo, Arild Waaler, Peter Haase, Ernesto Jiménez-Ruiz, Davide Lanti, Martin Rezk, Guohui Xiao, Özgür L. Özçep, and Riccardo Rosati. Optique: Zooming in on Big Data. *IEEE Computer*, 48(3):60–67, 2015.

[13] Steve Harris and Andy Seaborne. SPARQL 1.1 query language. W3C Recommendation, World Wide Web Consortium, March 2013. Available at http://www.w3.org/TR/sparql11-query.

[14] Roman Kontchakov, Martin Rezk, Mariano Rodriguez-Muro, Guohui Xiao, and Michael Zakharyaschev. Answering SPARQL queries over databases under OWL 2 QL entailment regime. In *Proc. of the 13th Int. Semantic Web Conf. (ISWC)*, volume 8796 of *Lecture Notes in Computer Science*, pages 552–567. Springer, 2014.

[15] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proc. of the 21st ACM Symp. on Principles of Database Systems (PODS)*, pages 233–246, 2002.

[16] Franck Michel, Loïc Djimenou, Catherine Faron-Zucker, and Johan Montagnat. Translation of relational and non-relational databases into RDF with xR2RML. In *Proc. of the 11th Int. Conf. on Web Information Systems and Technologies (WEBIST)*, pages 443–454, 2015.

[17] Boris Motik, Achille Fokoue, Ian Horrocks, Zhe Wu, Carsten Lutz, and Bernardo Cuenca Grau. OWL Web Ontology Language profiles. W3C Recommendation, World Wide Web Consortium, October 2009. Available at http://www.w3.org/TR/owl-profiles/.

[18] Kian Win Ong, Yannis Papakonstantinou, and Romain 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.

[19] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. on Database Systems*, 34(3):16:1–16:45, 2009.

[20] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *J. on Data Semantics*, 10:133–173, 2008.

[21] Mariano Rodriguez-Muro, Roman Kontchakov, and Michael Zakharyaschev. Ontology-based data access: Ontop of databases. In *Proc. of the 12th Int. Semantic Web Conf. (ISWC)*, volume 8218 of *Lecture Notes in Computer Science*, pages 558–573. Springer, 2013.

[22] Mary Tork Roth and Peter M. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proc. of the 23rd Int. Conf. on Very Large Data Bases (VLDB)*, pages 266–275. Morgan Kaufmann, 1997.

[23] Juan F. Sequeda, Marcelo Arenas, and Daniel P. Miranker. OBDA: Query rewriting or materialization? In practice, both! In *Proc. of the 13th Int. Semantic Web Conf. (ISWC)*, volume 8796 of *Lecture Notes in Computer Science*, pages 535–551. Springer, 2014.

[24] Jason Slepicka, Chengye Yin, Pedro A. Szekely, and Craig A. Knoblock. KR2RML: an alternative interpretation of R2RML for heterogenous sources. In *Proc. of the 6th Int. Workshop on Consuming Linked Data (COLD), co-located with ISWC*, volume 1426 of *CEUR Workshop Proceedings,* http://ceur-ws.org/, 2015.

[25] Jan Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science*, 254(1):363–377, 2001.

[26] Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, and Michael Zakharyaschev. Ontology-based data access: A survey. In *Proc. of the 27th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 5511–5519. IJCAI Org., 2018.