# A Generalized Framework for Ontology-based Data Access

Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Julien Corman, and Guohui Xiao

Faculty of Computer Science
Free University of Bozen-Bolzano, Italy
*lastname*@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.

## 1 Introduction

The database (DB) landscape has been significantly diversified during the last decade to satisfy the needs of a wide variety of modern applications. Traditional relational DB management systems now coexist with the 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 the following four main data models: the column-family, key-value, document, or graph data model, and while some of them can be queried through well-known declarative query languages, such as SQL or SPARQL, others offer ad-hoc querying mechanisms (e.g., the aggregation framework of MongoDB), or even require writing JavaScript functions (e.g., in 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 paper, we propose a generalization of the *ontology-based data access* (OBDA) framework as a uniform solution to this problem. The OBDA paradigm [22] 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, these queries are translated automatically into lower-level ones that DB engines can handle. This is done by exploiting 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

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

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 [10,5]. 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 [15], the mapping in specified in R2RML [7], and queries are formulated in SPARQL, the Semantic Web query language [11].

Our generalization extends the classical OBDA setting to arbitrary DBs. The approach enjoys all benefits already offered by OBDA, in particular hiding from the user low-level concerns such as data storage and access, and providing a high-level querying interface, closer to application needs. In the NoSQL case, an additional advantage is the possibility to formulate all queries in a familiar yet expressive language, namely SPARQL, which is both widespread and very similar to SQL dialects.

We then investigate the feasibility of the generalized OBDA framework when query answering is fully delegated to the DB engine, by focusing on MongoDB, a document-based DB, which is also one of the most popular NoSQL DBs as of today. OBDA appears as a solution of choice for MongoDB: MongoDB offers a very expressive query language, which however has a more procedural flavor than SQL or SPARQL, and can become very complex to manipulate for advanced information needs. Document-based DBs can leverage the *denormalized* structure of their data: a collection of documents can be seen as a materialized view over a (normalized) relational DB instance, essentially with joins being pre-computed. Therefore a natural question we try to answer in this paper is whether OBDA over MongoDB can take advantage of such structure in order to answer queries efficiently. We provide a first element towards a positive answer, by instantiating the generalized OBDA framework over MongoDB as an extension of the OBDA system *Ontop* [4], and comparing its performance with a triple store, which does not benefit from such denormalization.

The rest of the paper is structured as follows. Section 2 recalls the standard OBDA framework over relational data sources. Section 3 introduces our proposal for generalizing OBDA to access arbitrary DBs, and an architecture of a generalized OBDA system. Section 4 introduces MongoDB, and describes our extension of *Ontop* over MongoDB. Section 5 evaluates the performance of this system and compares it to the triple store Virtuoso, using as dataset an instance of the well-known Berlin SPARQL Benchmark (BSBM). Section 6 discusses related work, and Section 7 concludes the paper.

## 2 Ontology-based Data Access

We recall the traditional OBDA paradigm for accessing relational DBs through an ontology [22]. 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 $\mathcal{T}$ consists of axioms involving classes and properties, and does not mention individuals. In other words, $\mathcal{T}$ consists only of the intensional part of an ontology.

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. A *mapping assertion* [13] between $\mathcal{S}$ and $\mathcal{T}$ is an expression of the form

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

where $A$ is a class name in $\mathcal{T}$, $P$ is a 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. Mapping assertions allow one to define how classes and properties in $\mathcal{T}$ should be populated with objects constructed from values in a DB instance of $\mathcal{S}$.

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

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

where $\mathit{ans}(\varphi, D)$ denotes the result of the evaluation of $\varphi$ over $D$. Then a *model* of $\langle \mathcal{P}, D \rangle$ is simply a model of the ontology $\mathcal{T} \cup \mathcal{M}(D)$. We observe that in this ontology, $\mathcal{M}(D)$ provides the 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 [11,17]. Such queries are answered over the ontology $\mathcal{T} \cup \mathcal{M}(D)$, according to the semantics of the chosen *entailment regime*. Typically, in OBDA, the ontology $\mathcal{T}$ is expressed in OWL 2 QL, and the corresponding entailment regime is that of OWL 2 QL [12].

## 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 view $\mathbf{D}$-instances as flat relational DB instances: for a $\mathbf{D}$-instance $D$ satisfying a $\mathbf{D}$-schema $\mathcal{S}$, $[\![D]\!]$ is a flat relational DB over the relational schema $[\![\mathcal{S}]\!]$. The function $[\![\cdot]\!]$ is called *relational wrapper*.
- A *native* query language $\mathcal{Q}$, such that, for a query $q \in \mathcal{Q}$ and for a $\mathbf{D}$-instance $D$, the answer $\mathit{ans}(q, D)$ of $q$ over $D$ is defined, and is itself a $\mathbf{D}$-instance.

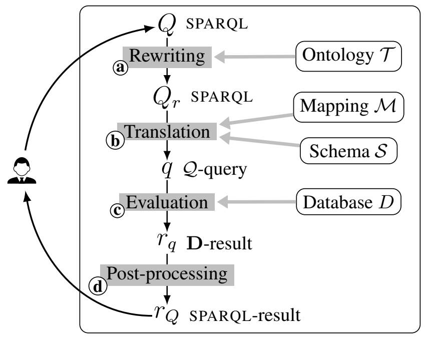

**Fig. 1.** Query Answering in OBDA

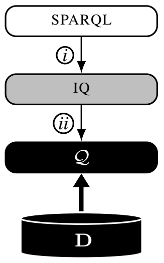

**Fig. 2.** SPARQL to native query translation

Now, given a $\mathbf{D}$-schema $\mathcal{S}$, a *mapping* $\mathcal{M}$ is a set of classical mapping assertions $\varphi \rightsquigarrow h$, where $\varphi$ is a SQL query over $[\![\mathcal{S}]\!]$. Then, as in the relational case, an *OBDA specification* is a triple $\langle \mathcal{T}, \mathcal{M}, \mathcal{S} \rangle$. 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 wrapper $[\![\cdot]\!]$.

Note that our assumption that for the class $\mathbf{D}$ of DBs a relational wrapper is available is not in any way restrictive, since any form of data, independently of how it is structured, can be represented using relations. 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, 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.

By default, when referring to OBDA, we mean the *virtual* approach, which avoids materializing the RDF graph, and instead delegates (part of) query answering to the DB. In this approach, the query answering process can be depicted as in Figure 1, and done in 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 regimes, this step only rewrites the basic graph pattern (BGPs) in $Q$ [12]). *(b)* The rewritten SPARQL query $Q_r$ is translated into one or several native queries $q \in \mathcal{Q}$. When the DB engine does not support (efficiently) some SPARQL operators, multiple native queries might be required, whose evaluation could be postponed to the final post-processing step. *(c)* The native queries $q$ are evaluated by the DB engine. *(d)* The results of all queries $q$ are combined and converted into SPARQL results 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 some new DB systems 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), it may be preferable to perform the unnesting (i.e., flattening) of nested objects into tuples as a post-processing step, so as to reduce network load between DB and client.

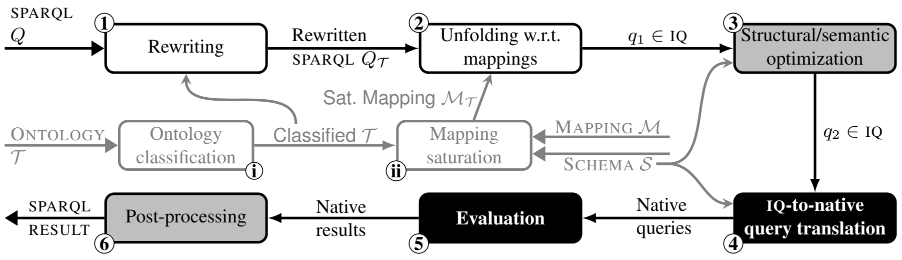

**Fig. 3.** Proposed architecture for an OBDA system

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 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 better supports optimizations 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 strictly 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 of an OBDA system able to answer SPARQL queries over heterogeneous data sources. This architecture, depicted in Figure 3, assumes an offline and an online query answering stages.

The *offline* stage (steps ⓘ and ⓘⓘ) takes as input the ontology, the mapping, and the schema, and produces two elements, to be used during the online stage: the *classified* ontology, and the *saturated* mapping [20,18], which is constructed by saturating the input mapping with the classified ontology. Notably, 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

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

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

**Fig. 4.** A collection of two MongoDB documents

native queries; ⑤ these are evaluated by the DB engine; 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 general, SPARQL queries are not aware of the structure of the stored data, hence the unfolded query may contain significantly more joins than necessary. 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.

## 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.

### 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, and vendor (in turn with id, name and homepage).

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

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

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

*product*

| nr | label |
|---|---|
| 23226 | Olympus OM-D E-M10 Mark II |
| 25887 | Panasonic Lumix DMC-GX80 |

*vendor*

| nr | label | homepage |
|---|---|---|
| 481 | amazon.it | http://www.amazon.it |
| 3785 | Yeppon Italia | http://www.yeppon.it |

*offer*

| nr | price | product | vendor |
|---|---|---|---|
| 258 | 747.14 | 23226 | 3785 |
| 311 | 500.32 | 25887 | 481 |
| 895 | 609.42 | 23226 | 481 |
| 922 | 759.99 | 23226 | 481 |

**Fig. 5.** Relational view of the collection in Figure 4, following the BSBM schema

BSBM benchmark [2], where the data is structured according to a relational schema consisting of multiple tables. Figure 5 provides the relational view corresponding to the above MongoDB collection, with distinct tables for products, offers, and vendors (the relational schema in the BSBM benchmark is actually more complex).

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 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 2 (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, given the value 481 for `offers.vendor.vendorId`, the value of `offers.vendor.name` associated to it can be retrieved from one document only. But in order to locate such data, MongoDB would check all documents with an occurrence of 481 for field `offers.vendor.vendorId`. Noticeably, this problem can be avoided by choosing a different document structure for the same data, with one document for each vendor rather than for each product, and consequently with redundant information about products. In general, the choice of a particular document structure is a trade-off, favoring some queries, and penalizing others. Thus, it should be done depending on 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 might need to be fetched.

MongoDB provides an ad-hoc querying mechanism for formulating expressive queries by means of the *aggregation framework*[4]. A *MongoDB aggregate query (*MAQ*)* is a sequence of *stages*, each of which takes one or two collections of documents as

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

```
db.bios.aggregate([
  {$project: {
     "productName": true, "offer1": "$offers", "offer2": "$offers" }},
  {$unwind: "$offer1"},
  {$unwind: "$offer2"},
  {$project: {
     "productName": true, "offer1": true, "offer2": true,
     "sameVendor": { $and: [
         {$ne: ["$offer1.offerId", "$offer2.offerId"]},
         {$eq: ["$offer1.vendorId", "$offer2.vendorId"]} ]}}},
  {$match: {"sameVendor": true} },
  {$project: {
     "productName": true, "vendorName": "$offer1.vendor.label",
     "price1": "$offer1.price", "price2": "$offer2.price" }}
])
```

**Fig. 6.** A MongoDB Aggregate Query (MAQ)

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

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

input, and produces another collection as output. A fragment of this language has been shown in [3] to be equivalent in expressive power to Nested Relational Algebra (NRA).

Because the language is powerful, MAQs can be complex to read and to manipulate. As an illustration, the MAQ of Figure 6 retrieves all products offered twice by the same vendor. In comparison, the SPARQL query of Figure 7 satisfies the same information need, but in a more concise fashion. The MAQ language also has a more procedural (less declarative) flavor than SQL/SPARQL, in 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 Extension of *Ontop* for MongoDB

We built a proof-of-concept prototype for answering SPARQL queries over MongoDB, called *Ontop/MongoDB*, which extends the *Ontop* system [4] 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,[5] thus minimizing the amount of post-processing required in step ⑥ of Figure 3. We now provide some details on the input accepted by *Ontop/MongoDB*, and on the implementation of steps ③ and ④.

*Ontop/MongoDB* takes as input an OWL 2 QL ontology, a mapping, and a set of constraints. In the current implementation, the source query $\varphi$ of each mapping assertion is relatively inexpressive: it can either retrieve a whole collection, or apply a simple filter to it. From the paths mentioned in this mapping, the system extracts the structure of the JSON documents being queried.

The constraints are user-defined functional dependencies that hold over the JSON documents being queried. For instance, in the collection of Figure 4, the value of path `offers.vendor.vendorID` uniquely defines the values of paths `offers.vendor.name` and `offers.vendor.homepage`. Note that, although MongoDB does not enforce such constraints, they can still hold over the data, and can be used for semantic optimization, e.g., to eliminate redundant joins. Note also that if the JSON collection is a denormalized version of an existing relational DB instance (as is the case for BSBM), then such dependencies can be directly inferred from keys declared in the relational schema.

In step ③, in addition to 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 constraints just mentioned. In step ④, *Ontop/MongoDB* uses an optimized version of the NRA-to-MAQ translation given in [3], which in theory makes full delegation of query answering to MongoDB feasible, but, if left unoptimized, may produce queries not executable in practice. An important consideration here is the internal limitations put by MongoDB on the size of in-memory intermediate results during query evaluation (16 MB for a single document, and 100 MB for a collection). Another purpose of this optimization is to take advantage of indexes available over the source JSON collection.

## 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 leverage 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 [9] representing 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)

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

triples sharing a constant value[6]. Comparatively, 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. On the other hand, 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)* that *Ontop/MongoDB* outperforms Virtuoso on queries containing a unique constant in an indexed field and fetching a single document; *(ii)* that Virtuoso outperforms *Ontop/MongoDB* on queries containing only constants with 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 [2], 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 query, replacing constant placeholders with values randomly sampled from the data. One version of each query was set aside for a cold run, and the $8 \cdot 9$ remaining 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 run 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.[7]

[6] http://docs.openlinksw.com/virtuoso/rdfperfrdfscheme/

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

**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 |

### 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 cross-document operations are only used for joins that cannot be performed within each document.

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 4.2). 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 ($\leq 5$ 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 contains 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 [19]. In recent years, several practical systems were developed for querying MongoDB via SQL: Drill[8], Dremio[9], Studio 3T[10], and the MongoDB Connector for Spark[11]. 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.

Another line of relevant research is the SQL++ extension of the SQL language for accessing, e.g., JSON data [16]. SQL++ has been supported by the DB engines Couchbase[12] and AsterixDB[13].

There already exist several mapping language proposals extending R2RML for converting non-relational data sources to RDF, e.g., RML [8], xR2RML [14], KR2RML [21], and D2RML [6]. These languages extend the relational model used in R2RML to more general cases (e.g., CSV, JSON, and Web Services). Their corresponding systems are mostly used for data conversion; the xR2RML implementation also supports SPARQL query answering by partially materializing the relevant RDF graph.

Finally, the approach of [1] 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.

## 7 Conclusions

In this paper, 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 query languages provided by native NoSQL DBs. We also propose a practical architecture for a generalized virtual OBDA approach, that allows one to answer 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 have compared its performance to that of a triple store. The evaluation we have carried out shows that *Ontop/MongoDB* was able to generate MAQs 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 MAQ, using a wider range of queries, but also different document structures for the same dataset.

[8] `https://drill.apache.org/`
[9] `https://www.dremio.com/`
[10] `https://studio3t.com/whats-new/how-to-query-mongodb-with-sql/`
[11] `https://docs.mongodb.com/spark-connector/`
[12] `http://couchbase.com/`
[13] `https://asterixdb.apache.org/`

## References

1. Araujo, T.H.D., Agena, B.T., Braghetto, K.R., Wassermann, R.: OntoMongo – Ontology-Based data access for NoSQL. In: Proc. OntoBras. CEUR, `ceur-ws.org`, vol. 1908 (2017)
2. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. Int. J. Semantic Web and Information Systems **5**(2), 1–24 (2009)
3. Botoeva, E., Calvanese, D., Cogrel, B., Xiao, G.: Expressivity and complexity of MongoDB queries. In: Proc. ICDT. LIPIcs, vol. 98, pp. 9:1–9:22 (2018)
4. Calvanese, D., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., Xiao, G.: Ontop: Answering SPARQL queries over relational databases. Semantic Web J. **8**(3), 471–487 (2017)
5. Calvanese, D., Liuzzo, P., Mosca, A., Remesal, J., Rezk, M., Rull, G.: Ontology-based data integration in EPNet: Production and distribution of food during the Roman Empire. Engineering Applications of Artificial Intelligence **51**, 212–229 (2016)
6. Chortaras, A., Stamou, G.: D2RML: Integrating heterogeneous data and web services into custom RDF graphs. In: Proc. LDOW. CEUR, `ceur-ws.org`, vol. 2073 (2018)
7. Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF mapping language. W3C Rec. (2012), available at `http://www.w3.org/TR/r2rml/`
8. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: A generic language for integrated RDF mappings of heterogeneous data. In: Proc. LDOW. CEUR, `ceur-ws.org`, vol. 1184 (2014)
9. Erling, O., Mikhailov, I.: RDF support in the Virtuoso DBMS. In: Networked Knowledge – Networked Media, Studies in Computational Intelligence, vol. 221. Springer (2009)
10. Giese, M., Soylu, A., Vega-Gorgojo, G., Waaler, A., Haase, P., Jiménez-Ruiz, E., Lanti, D., Rezk, M., Xiao, G., Özçep, Ö.L., Rosati, R.: Optique: Zooming in on Big Data. IEEE Computer **48**(3), 60–67 (2015)
11. Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C Rec. (2013), available at `http://www.w3.org/TR/sparql11-query`
12. Kontchakov, R., Rezk, M., Rodriguez-Muro, M., Xiao, G., Zakharyaschev, M.: Answering SPARQL queries over databases under OWL 2 QL entailment regime. In: Proc. ISWC. LNCS, vol. 8796, pp. 552–567. Springer (2014)
13. Lenzerini, M.: Data integration: A theoretical perspective. In: Proc. PODS (2002)
14. Michel, F., Djimenou, L., Faron-Zucker, C., Montagnat, J.: Translation of relational and non-relational databases into RDF with xR2RML. In: Proc. WEBIST. pp. 443–454 (2015)
15. Motik, B., Fokoue, A., Horrocks, I., Wu, Z., Lutz, C., Cuenca Grau, B.: OWL Web Ontology Language profiles. W3C Rec. (2009), available at `http://www.w3.org/TR/owl-profiles/`
16. Ong, K.W., Papakonstantinou, Y., Vernoux, R.: The SQL++ query language: Configurable, unifying and semi-structured. CoRR Technical Report abs/1405.3631, arXiv.org (2014)
17. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM TODS **34**(3), 16:1–16:45 (2009)
18. Rodriguez-Muro, M., Kontchakov, R., Zakharyaschev, M.: Ontology-based data access: Ontop of databases. In: Proc. ISWC. LNCS, vol. 8218, pp. 558–573. Springer (2013)
19. Roth, M.T., Schwarz, P.M.: Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In: Proc. VLDB. pp. 266–275. Morgan Kaufmann (1997)
20. Sequeda, J.F., Arenas, M., Miranker, D.P.: OBDA: Query rewriting or materialization? In practice, both! In: Proc. ISWC. LNCS, vol. 8796, pp. 535–551. Springer (2014)
21. Slepicka, J., Yin, C., Szekely, P.A., Knoblock, C.A.: 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. CEUR, `ceur-ws.org`, vol. 1426 (2015)
22. Xiao, G., Calvanese, D., Kontchakov, R., Lembo, D., Poggi, A., Rosati, R., Zakharyaschev, M.: Ontology-based data access: A survey. In: Proc. IJCAI. AAAI Press (2018)