OntoMongo - Ontology-Based Data Access for NoSQL

Thiago H. D. Araujo[1], **Barbara T. Agena**[1], **Kelly R. Braghetto**[1], **Renata Wassermann**[1]

[1]Instituto de Matemática e Estatística – Universidade de São Paulo

{thiagohd,bagena,renata,kellyrb}@ime.usp.br

Abstract. *Ontology-based data access (OBDA) has gained attention in recent years for providing access to large volumes of data by using ontologies as a conceptual layer and exploring their ability to describe domains and deal with data incompleteness. This is done through mappings that connect the data in the database to the vocabulary of the ontology. The first OBDA studies were about data stored in relational databases. Recent studies have begun to extend the use of OBDA to NoSQL databases. In this paper, we present a novel approach for OBDA with document-oriented NoSQL databases. Differently from related works, our approach uses an access interface with an intermediate conceptual layer that is extensible and flexible, capable of providing access to different types of database management systems. To validate the approach, we have implemented a prototype for MongoDB, using a real-world application domain as a case study.*

1. Introduction

In the past, large corporations were interested in collecting large volumes of data. Nowadays, with the emergence of Big Data, this task became rather easy, the new challenge is to mine and extract useful and valuable information from these large data collections. People need technical knowledge and domain knowledge in order to design relevant queries and do the analysis. Calvanese et al. (Calvanese et al. 2007) proposed a new solution to the problem of developing simple and effective query methods by using ontologies, the Ontology-Based Data Access (OBDA).

OBDA works by having an ontology that describes a certain domain, a relational database for data storage and a mapping, connecting the vocabulary of the ontology to the sets of data present in the database. It also employs a query-answering method that uses the mapping to access and transform data in an efficient manner. SPARQL queries are rewritten as SQL language queries. The result of the SQL queries is transformed into ontology instances, so it's easy to infer new facts and build new formal knowledge by using the ontology. Building a conceptual layer that is independent of the underlying structure of the database was shown to be efficient (Botoeva et al. 2016b) when you have users that know a lot about the domain but don't have any knowledge about how the data is organized and stored.

There are scenarios where the amount of available data grows exponentially, the variety of data formats is large, and there's a need to store and analyze semi-structured or even unstructured data, and relational databases are not the best

solution for storage and querying in these scenarios (Nayak et al. 2013). NoSQL (Not Only SQL) databases were developed in order to provide better performance, availability, and flexibility.

There are four main groups of NoSQL Database Management System (DBMS): document-oriented, column-oriented, graph-oriented and key-value stores. As better explained in (Nayak et al. 2013), document-oriented databases are schemaless and they store data as documents like XML and JSON, offering very good performance and scalability. *MongoDB*[1]is a very popular document-oriented DBMS. Column-oriented databases are similar to relational databases but are optimized for reading and writing columns of data instead of rows, and they are very good for analytic queries. Graph-oriented databases represent information as nodes and edges on a graph, and they are good for representing associations between things or concepts as a network. Key-value stores are pretty much like a large hash table, and they are good for fast data retrieval (reads) and caching.

With the increasing popularity of NoSQL DBMSs, there is growing interest in applying OBDA to these types of systems. But unlike relational DBMSs, there are neither standard query languages nor data models for NoSQL DBMSs. In this work, we present an approach for the use of OBDA with NoSQL Systems, so that it is possible to apply ODBA in domains with large volumes of data. We also aim to simplify the mapping and the process of converting SPARQL queries into queries for different types of NoSQL systems. To do this, we propose an intermediate conceptual layer that represents the models saved in the database and an extensible method of translating the queries to the target DBMSs. To evaluate its viability, we apply this approach in an ODBA that uses a document-oriented NoSQL database in MongoDB.

2. Background

Botoeva et al. (Botoeva et al. 2016b) proposed the use of NoSQL in OBDA applications. Their work is an extension of Ontop (Bagosi et al. 2014), which is an OBDA system for relational databases. The new proposed architecture was tested using a document-oriented MongoDB database. In a related work, Botoeva et al. (Botoeva et al. 2016a) presented a formal evaluation of a subset of data access queries available on MongoDB. This evaluation has shown that it is very hard to build a completely generic framework that is able to query any kind of NoSQL DBMS. Unlike relational databases that use a common query language (SQL), NoSQL DMBS share few query patterns, so every NoSQL DBMS needs a specific query translator.

In another interesting work, Michel et al. (Michel et al. 2016) have built a SPARQL to MongoDB query mapping tool in order to turn a legacy database into a publicly available data source. The database should be exposed as a virtual RDF database: stored documents would be published as RDF triples. The tool does the translation in two steps: first, the SPARQL query is transformed into an abstract query using mappings from MongoDB documents to RDF written in an intermediate language called xR2RML. Afterward, the query is rewritten to a concrete MongoDB

[1]MongoDB `https://www.mongodb.com/`

query. As a result, they've shown that it is always possible to rewrite a query that produces correct results.

3. ODBA with NoSQL Databases

In this section, we talk about our approach to the problem of using ODBA to access NoSQL databases.

3.1. Query Translation Method

We propose the utilization of an OBDA model with an *Access Interface* comprised of an ontology, a mapping and an intermediate conceptual layer capable of providing data access to a NoSQL DBMS. Our approach aims to simplify the construction of a mapping to make it more flexible and generic, allowing its reuse independently of the DBMS selected for the persistence layer. In order to do that, we define an intermediate conceptual layer using classes in an object-oriented programming (OOP) language to represent the structure (schema) of the data persisted on the database.

On the one hand, this approach simplifies the construction of a mapping between ontology and OOP classes, since an object-oriented representation can be very close to the ontological domain representation. Knowledge representation in an ontology can be defined via classes, relations, hierarchy and inheritance, ideas very similar to the concepts behind an object-oriented language implementation. On the other hand, the task of accessing the data persisted in databases and mapping in objects can be delegated to ORM (Object-relational Mapping) or ODM (Object Document Mapper) libraries, very frequently used in industry. These libraries can access several types of data stored in different DBMS, such as relational databases or document-oriented databases, and turn them into object collections.

In the mapping proposed here, each class (or attribute) of the ontology must be associated with its respective class (or attribute) in the OOP model. In turn, the OOP model is associated with the database schema by means of an ORM or ODM. Once this mapping is defined, the Access Interface is able to receive a SPARQL query and perform the data recovery. From the SPARQL query, a graph describing the data which will be accessed by the query in the database is generated. The nodes in the graph represent the classes, while the edges denote the relations between the classes. Each node contains the list of attributes of the correspondent class required to answer the SPARQL query.

The description of all data that must be recovered from the database in order to answer a SPARQL query is codified in the mapping and in the graph. In this way, the data can be recovered by means of data access operations available in the ORM/ODM library or by internal mechanisms which generate queries for the employed DBMS.

A first implementation of this method – a project called *onto-mongo* – targeted document-oriented NoSQL DBMS; it is described in Section 3.2.

3.2. The onto-mongo Project

The *onto-mongo*[2] project is a working prototype that applies the query translation method detailed in Section 3.1, generating a set of RDF triples as a result that is

then used to populate an OWL ontology. As illustrated in Figure 1, the *onto-mongo* project has the following main components: an OWL Ontology, a NoSQL database, an Access Interface, an Ontology to Database collections mapping, a SPARQL to NoSQL query translator, and a RDF export.

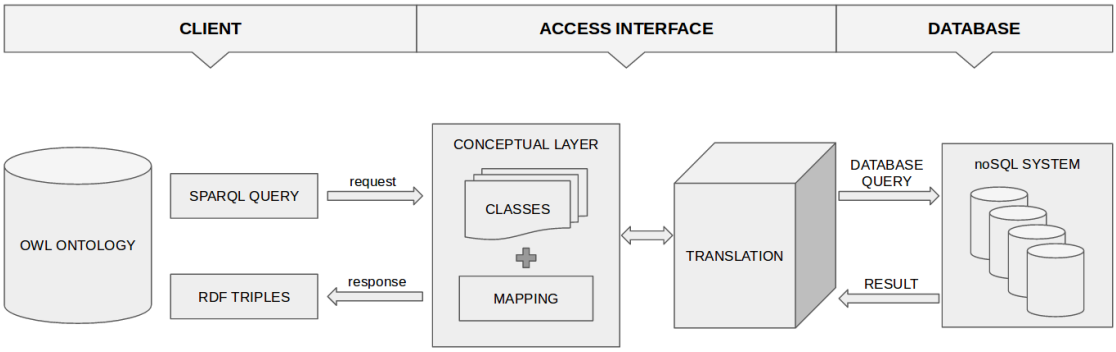

Figure 1. Architecture of the *onto-mongo* project

3.2.1. OWL Ontology

As an example, an ontology called *basic-lattes* (Figure 2), about researchers and their publications, was created with the ontological classes *Researcher* and *Publication*. The ontology is a client to our application, and it is used to answer some competency questions related to the domain, which can be expressed as SPARQL queries.

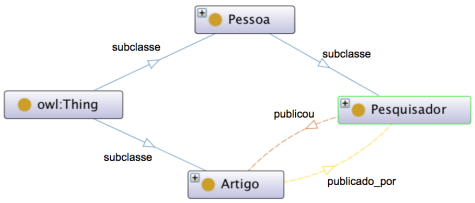

Figure 2. *basic-lattes* ontology

3.2.2. NoSQL database

The *onto-mongo* project uses a document-oriented NoSQL database as data storage. As an example of application, we have used data about researchers' scientific production, extracted from the Lattes Platform[3] using a tool called *scriptLattes* (Mena-Chalco and Cesar Junior 2009). This tool is able to crawl and export researchers' Academic CVs as XML files which contain very detailed information about scientific publications. After that, we have converted the XML files into *JSON* documents and selected some useful fields like name, article name, and coauthors. We

[2]The project is open-source: `http://github.com/thdaraujo/onto-mongo`

have stored about a hundred of these documents into a document-oriented NoSQL database maintained in the MongoDB DBMS.

3.2.3. Access Interface

The Access Interface is a module capable of transforming a message containing a SPARQL query and answering the message with a set of RDF triples extracted from the database. This interface was developed using *ruby*[4], an object-oriented language, while the *ruby on rails*[5] web framework was used to create a service application to receive the messages.

The Access Interface has classes written in *ruby* which represent the structure of the database-stored collections. We call them *model* classes. Our example has two main model classes: *Researcher* and *Publication*. The former models data from researchers, and the latter models data from scientific publications.

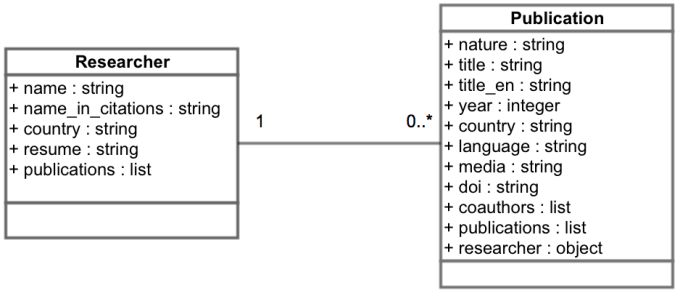

Figure 3. Model classes diagram: *Researcher* and *Publication*

It is possible to define all of the instance attributes which are persisted in the database by adding annotations to the model classes (i.e.: *name* or *country*). It's also possible to define the relations between classes, indicating for example that an instance of the model Researcher might have a list of his or her publications that is also saved in the database. And, on the other side, we can indicate that a *Publication* must be related to an instance of Researcher.

Model classes are used to manipulate persistent objects. They interact with the database by means of CRUD (create, read, update, delete) methods. When a NoSQL database is used to persist the objects, the model classes can be seen as a conceptual schema for the database. For this reason, the model classes facilitate the ontology mapping phase.

[3]The Lattes Platform is a system maintained by the Brazilian Government to manage information on science, technology, and innovation related to researchers working in Brazil.

[5]The Ruby Language `https://www.ruby-lang.org/`

[5]Ruby on Rails `http://rubyonrails.org/`

3.2.4. Mapping

The mapping links the ontology classes to the Access Interface classes so that it is possible to query information related to these entities in the database. Using this approach, an ontology class is mapped to a ruby class that represents similar concepts. The relationships between classes of the ontology are also mapped to relations between classes of the object-oriented language.

The mapping is done through the creation of methods that indicates the equivalences that exist between the ontology and the model classes. Thus, it became possible to map the structure present in the ontology to the structure of the data collections present in the database, passing through the classes of the Access Interface. The mapping can be defined in a script placed in the *initialize* folder of the rails project, which will be executed when the program start up. In this script, the user can define the mapping for each class of the Ontology, telling which class model represents the class, which attributes of this model are equivalent to the attributes of the ontology class, and which relations between the collections in the document-oriented database are equivalent to the relations between the ontology classes. In the mapping, the user can also indicate the path of the OWL file which contains the definition of the ontology.

As an example, consider the following mapping for the ontology class Researcher:

Listing 1. Mapping of Researcher class to Researcher model

```
1 OntoMap.mapping 'Pesquisador' do
2   model Researcher
3   maps from: 'nome', to: :name
4   maps from: 'pais', to: :country
5   maps from: 'nome_em_citacoes', to: :name_in_citations
6   maps relation: 'publicou', to: :publications
7 end
```

In line 1, we call the method *mapping* of the OntoMap module, passing as parameter the class **Researcher** present in the ontology, and a block with some definitions. In line 2, we say that the *model* **Researcher** present in the interface is equivalent to the Researcher class. In lines 3 to 5, we map some attributes of the model to attributes of the class in the ontology. The *maps* method receives two parameters: *from*, which refers to some relation or property belonging to an ontology class and *to*, that indicates which attribute of the class *Researcher* or relation with another class is equivalent. In line 3, we define a mapping between the property *name* and the attribute *name*. In line 6, the relation *published* that exists between the classes Researcher and Publication is mapped to the relation *publications* of the model, which represents the set of publications of a given researcher. It is important to note that an instance of the class Researcher may have one or more publications. Similarly, the mapping between the class Publication and the model Publication can be made as follows:

Listing 2. Mapping publication class to Publication model

```
1 OntoMap.mapping 'Publicacao' do
```

```
2    model Publication
3    maps from: 'natureza', to: :nature
4    maps from: 'titulo', to: :title
5    maps from: 'titulo_em_ingles', to: :title_en
6    maps from: 'ano', to: :year
7    maps from: 'pais', to: :country
8    maps from: 'idioma', to: language
9    maps from: 'veiculo', to: media
10   maps from: 'doi', to: doi
11 end
```

The presented mapping allowed the translation of SPARQL queries into MongoDB queries without the need for a direct mapping between the ontology structure and the database document collections. In other related works that addressed OBDA for relational DBMS, this type of mapping is done directly between SQL queries and ontology classes. Our approach thus has a conceptual layer serving as a link between ontology and data collections, and this conceptual layer (*model*) is described in an object-oriented language, following a software architectural pattern widely used. One of the main advantages of this approach is the possibility of reusing the conceptual layer for other types of DBMS. With no efforts, we can use this layer to persist and access data in relational databases.

3.2.5. Translator

The translator uses the mapping to transform SPARQL queries into queries to the DBMS. The SPARQL query is first transformed into a *SPARQL Syntax Expression* (S-Expression) through the *sxp* library [6], as this facilitates translation. The query is then transformed into a graph representing the RDF triples, filters, groupings, and variables that must be returned for this query. The translator has two different strategies: the first one translates the query using the data recovery methods from the model classes, and the other generates a MongoDB query in JSON, with aggregations and filters, which is executed by the DBMS itself. The first strategy is simpler because it only transforms the query into calls to methods of the model classes along with Mongoid filtering methods, such as the *WHERE* method, which receives a set of filters that must be applied to a list of objects. The retrieved data objects can also be processed with the filtering and map/reduce functions of the ruby collection libraries. The second strategy is the most appropriate for complex queries on large databases, which have several types of aggregations, or filtering for internal or related collections. With this strategy, these operations are executed by the database management system, which provides a better performance.

Currently, the translator only supports SPARQL queries in which the predicate of the RDF triples in the *WHERE* clauses are relations or attributes defined in the ontology and equality filters. The query in Listing 3 is an example of query supported by the translator.

[6] Available at: `https://github.com/dryruby/sxp.rb`

3.2.6. RDF export

The sets of objects extracted from the database are transformed into RDF triples and sent back to the initial caller as the answer. We are also able to insert these triples into the ontology as a way to populate it. Afterward, this information can be queried or analyzed directly without any prior knowledge about the structure of the database collections or the type of NoSQL system that is being used.

4. Case study

A case study was created to evaluate the feasibility of our approach, and we will discuss it as an example to the reader. By starting with a query very similar to the one described by Botoeva et al. (Botoeva et al. 2016a), we want to know which researchers have published two different publications in the same year. To answer that, we needed to implement only the subset of the NoSQL query language that deals with aggregation. We also used a document-oriented database.

The NoSQL database contains about a hundred CVs extracted from the Lattes Platform. We also added some information about the researcher Kristen Nygaard and two of his publications that have the same year so that our example would be easier to illustrate. This was added to the *researchers* collection:

- "COOL (comprehensive object-oriented learning)", 2002.
- "Class and Subclass Declarations", 2002.
- "Classification of actions and inheritance also for methods", 1987.

Using the same ontology discussed at section 3.2.1, we need to answer the following question: *which researchers published two different publications in the same year?* We can ask *onto-mongo* this question expressed in the following SPARQL query:

Listing 3. Example SPARQL query

```
PREFIX : <http://onto-mongo/basic-lattes/#>
SELECT
    ?nomePesquisador ?tituloPublicacao1 ?tituloPublicacao2 ?ano
WHERE
{
    ?pesquisador :nome ?nomePesquisador .
    ?pesquisador :publicou ?publication1 .
    ?pesquisador :publicou ?publication2 .
    ?publicacao1 :ano_publicacao ?ano .
    ?publicacao2 :ano_publicacao ?ano .
    ?publicacao1 :titulo ?tituloPublicacao1 .
    ?publicacao2 :titulo ?tituloPublicacao2 .
    FILTER
        (?publicacao1 != ?publicacao2)
}
```

First, we transform this query into an equivalent SPARQL Syntax Expression, using the *sxp* library:

Listing 4. SPARQL query transformed into a S-Expression

```
1 (project
```

```
2    (?nomePesquisador ?tituloPublicacao1 ?tituloPublicacao2 ?ano)
3    (filter
4      (!= ?publicacao1 ?publicacao2)
5      (bgp
6        (triple ?pesquisador :nome ?nomePesquisador)
7        (triple ?pesquisador :publicou ?publicacao1)
8        (triple ?pesquisador :publicou ?publicacao2)
9        (triple ?publicacao1 :ano_publicacao ?ano)
10       (triple ?publicacao2 :ano_publicacao ?ano)
11       (triple ?publicacao1 :titulo ?tituloPublicacao1)
12       (triple ?publicacao2 :titulo ?tituloPublicacao2))))
```

The expression is divided into its components so that they can be translated to a database query. The algorithm divides the query into the following components: output variables (*outputs*), filters that must be applied (*filters*), the query body (*body*) and its attribute triples (*attributes*) and relations (*relations*). We will discuss these components in the next section by their order of evaluation.

4.1. Query Body

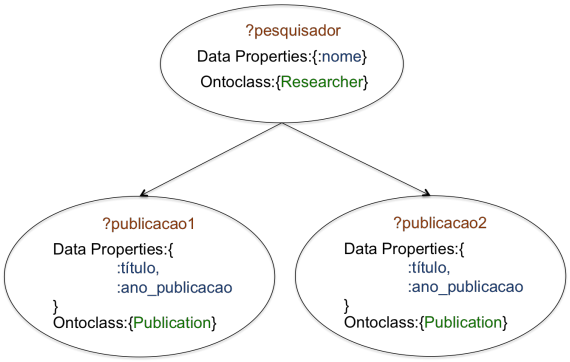

Figure 4. Graph built from the SPARQL query

From the SPARQL query, we construct the graph of Figure 4. In line 6 of this query example, the first triple is read and the first node of the graph of Figure 4 is generated containing the attribute *:nome* of class *Pesquisador* stating that the name of the researcher must be retrieved in the search that will be executed in MongoDB. From the mapping, we know that the field : *nome* is related to the field *name* present in *Researcher*, that is, it is just an **attribute**. In MongoDB, we say that this field should be projected (*projection*). *Project* is a command that returns JSON fields in the resulting document. It is similar to columns returned in a *SELECT* clause in SQL. Then, in lines 7 and 8 we create two nodes that are adjacent to the first node, referring to class *Publicação*, because there is a relation *publicou*. We know from the mapping that *publicou* is a **relation** between *researcher* and *publication* and *publication* is a list. For this reason, we generate an *unwind* command of *publicacao*1 and another of *publicacao*2. The *unwind* command output a document for each element of a list field. Because MongoDB does not have the JOIN operation, it is necessary to transform a list or array present within the document into a set of other objects for each element of that list using *unwind*. In the case of our example, the combined effect of the two *unwind* is equivalent to a SQL cross join between *publicacao*1 and *publicacao*2.

Since *publication* is a document that is inside the document *researcher*, we add the prefix so its attributes *year* and *title* can be found. Therefore, we make the projection of these attributes with its prefix(e.g.: ?*publicacao*1.*year*). Once the *projections* and *unwinds* are made, we then analyze the filters.

4.2. Filters

In lines 3 and 4 of the query in Listing 4, we can see the filters that should be applied to the query result. In this case, ?*publicacao*1 must be different from ?*publicacao*2. But we also take into account the information in lines 9 and 10: the query requires that the year of the publications be the same, since variable ?*ano* appears in ?*publicacao*1 and ?*publicacao*2. Hence, the filter should ensure that each of the publication pairs has different values for all their fields except by the year, which should be the same. In this case, the query should bring in all the distinct publications that were published in the same year.

4.3. Output

The output variables can be seen on line 2. When the output projection is generated, it's important to add the prefixes that are needed and then we are able to link the variables and the attributes present in the document by using the mapping. As an example, we can see that ?*tituloPublicacao*1 gets its value from the field $*publicacao*1.*title* that has been projected.

4.4. Final Query and Results

The resulting query that is executed is the following:

Listing 5. Resulting query

```
{ :$project =>
    { :name => true,
      'publicacao1' => '$publications',
      'publicacao2' => '$publications' } },
 { :$unwind => '$publicacao1' },
 { :$unwind => '$publicacao2' },
 { :$project =>
    { :name => true,
      :publicacao1 => true,
      :publicacao2 => true,
      'filter_1' =>
      { '$and' =>
        [{ :$eq => ['$publicacao1.year', '$publicacao2.year'] },
         { :$ne => ['$publicacao1.title', '$publicacao2.title']
           ↪ }] } } },
 { :$match => { 'filter_1' => true } },
 { :$project =>
    { 'nomePesquisador' => '$name',
      'tituloPublicacao1' => '$publicacao1.title',
      'tituloPublicacao2' => '$publicacao2.title',
      'ano' => '$publicacao1.year' } }
```

By querying the sample data, we get as a result a JSON document containing unique publications that were published in the same year and that have the same author, which answers the initial question:

Listing 6. JSON documents that answer the query

```
[{ '_id' => BSON::ObjectId('5835effcc048bd0001cdc239'),
   'researcherName' => 'Kristen Nygaard',
   'publicationTitle1' => 'COOL (Comprehensive Object-oriented
      ↪ Learning)',
   'publicationTitle2' => 'Class and Subclass Declarations',
   'year' => 2002 }]
```

These documents are transformed into triples and then inserted on a RDF graph, which is returned as an answer to the initial query. It's also possible to insert this new knowledge into the ontology and run some other queries or apply any form of logical inference.

5. Related and future work

The approach presented in this research offers advantages for the scientific community, mainly because it provides a different way of constructing mappings, making this process easier to OBDA Systems.

By avoiding a direct mapping to the database query language, as done by (Calvanese et al. 2017), the effort to construct and maintain the mapping is reduced. Unlike the approach to relational DBMSs whose queries are written in standard SQL language, each NoSQL DBMSs have different and diverse query languages.

Placing next to the mapping a conceptual layer that operates on classes in an OOP language makes our approach a more generic and flexible form of data access.

In the future, we intend to analyze the differences between our approach and existing approaches, evaluating advantages and disadvantages and investigating the amount of effort required to apply this solution to other types of NoSQL DBMS.

A future extension may allow **onto-mongo** to access other types of NoSQL DBMSs, such as Cassandra. We also need to make the translator more generic and flexible, as well as provide support to other types of queries like *joins*, filters, and groupings. A possible solution is to develop an abstract interface describing the methods that need to be implemented for a specific query translator, and then an intermediate user (the user who configured the mapping) will be able to implement these methods for a different kind of DBMS the user needs.

Also, we intend to simplify the mapping method so that the user can create it with as little knowledge of the Ruby language as possible.

One application of our approach currently being developed is a project to handle data extraction from the Lattes Platform in order to find co-authorship networks via Link Prediction and Machine Learning algorithms. For that, **onto-mongo** is being used to extract semantic information from a NoSQL database through SPARQL queries.

Another application in development by our research group will use this approach to access data from a MongoDB database on legal decisions.

References

[Bagosi et al. 2014] Bagosi, T., Calvanese, D., Hardi, J., Komla-Ebri, S., Lanti, D., Rezk, M., Rodríguez-Muro, M., Slusnys, M., and Xiao, G. (2014). The Ontop Framework for Ontology Based Data Access. In *8th Chinese Semantic Web and Web Science Conference*, pages 67–77.

[Botoeva et al. 2016a] Botoeva, E., Calvanese, D., Cogrel, B., Rezk, M., and Xiao, G. (2016a). A Formal Presentation of MongoDB (Extended Version). *CoRR Technical Report abs/1603.09291, arXiv.org e-Print archive, 2016*.

[Botoeva et al. 2016b] Botoeva, E., Calvanese, D., Cogrel, B., Rezk, M., and Xiao, G. (2016b). OBDA Beyond Relational DBs : A Study for MongoDB. *Proc. of the 29th Int. Workshop on Description Logics (DL 2016)*, 1.

[Calvanese et al. 2017] Calvanese, D., Cogrel, B., Komla-ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., and Xiao, G. (2017). Ontop : Answering SPARQL Queries over Relational Databases. *Semantic Web Journal*, 8(3):471–487.

[Calvanese et al. 2007] Calvanese, D., Giacomo, G. D., Lembo, D., Lenzerini, M., Poggi, A., and Rosati, R. (2007). Ontology-based database access. In *Proc. of the 15th Italian Symposium on Advanced Database Systems(SEBD'07)*, pages 324–331.

[Mena-Chalco and Cesar Junior 2009] Mena-Chalco, J. A. P. and Cesar Junior, R. M. (2009). ScriptLattes: an open-source knowledge extraction system from the Lattes platform. *Journal of the Brazilian Computer Society*, 15:31 – 39.

[Michel et al. 2016] Michel, F., Faron-Zucker, C., and Montagnat, J. (2016). A mapping-based method to query MongoDB documents with SPARQL. In *International Conference on Database and Expert Systems Applications*, pages 52–67. Springer.

[Nayak et al. 2013] Nayak, A., Poriya, A., and Poojary, D. (2013). Types of NOSQL Databases and its Comparison with Relational Databases. *International Journal of Applied Information Systems*, 5(4):16–19.