View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by HAL-Ecole des Ponts ParisTech

On the Potential Integration of an Ontology-Based Data Access Approach in NoSQL Stores

Olivier Curé, Fadhela Kerdjoudj, David Célestin Faye, Chan Le Duc, Myriam Lamolle

▶ To cite this version:

Olivier Curé, Fadhela Kerdjoudj, David Célestin Faye, Chan Le Duc, Myriam Lamolle. On the Potential Integration of an Ontology-Based Data Access Approach in NoSQL Stores. IEEE Computer Society. Emerging Intelligent Data and Web Technologies, Sep 2012, Bucarest, Romania. pp.166-173, 2012. <hal-00799030>

HAL Id: hal-00799030

https://hal-upec-upem.archives-ouvertes.fr/hal-00799030

Submitted on 11 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On The Potential Integration of an Ontology-Based Data Access Approach in NoSQL Stores

Olivier Curé, Fadhela Kerdjoudj
Université Paris-Est Marne-la-Vallée, LIGM, France
{*ocure,f.kerdjoudj*}*@univ-mlv.fr*
David Faye
Université Gaston Berger de Saint-Louis, Sénégal
david-celestin.faye@ugb.edu.sn

Chan Le Duc, Myriam Lamolle
LIASD
Université Paris 8 Montreuil, France
{*chan.leduc,myriam.lamolle*}*@iut.univ-paris8.fr*

***Abstract*—NoSQL stores are emerging as an efficient alternative to relational database management systems in the context of big data. Many actors in this domain consider that to gain a wider adoption, several extensions have to be integrated. Some of them focus on the ways of proposing more schema, supporting adapted declarative query languages and providing integrity constraints in order to control data consistency and enhance data quality. We consider that these issues can be dealt with in the context of Ontology Based Data Access (OBDA). OBDA is a new data management paradigm that exploits the semantic knowledge represented in ontologies when querying data stored in a database. We provide a proof of concept of OBDA's ability to tackle these three issues in a social application related to the medical domain.**

***Keywords*-Ontology Based Data Access (OBDA); NoSQL; Document store; SPARQL; Social Application**

I. INTRODUCTION

NoSQL covers a wide range of technologies and data architectures for managing web-scale data and having the following common features: persistent data, non-relational data, avoid join operations , distribution, massive horizontal scaling, no fixed and flexible schemata, replication support, individual usually procedural query systems rather than using a standard declarative query language, consistent within a node of the cluster and eventually consistent across the cluster and simple transactions. According to their data model and replication/sharding strategy, we distinguish four NoSQL categories, each one having its own specificities and facilitating the management of some particular kind of data: view of a database as something for storing a value (Key-value Stores), more flexibility about stored data (Document Stores), management of use cases like relationships (Graph Databases) or aggregation of data (Column Databases).

Solutions in the NoSQL ecosystem are emerging in various domains such as social, scientific and even financial applications. Nevertheless, many actors consider that in order to increase its adoption rate, NoSQL systems need to integrate some new features. In fact, the desired features correspond to the ones found in Relational Database Management Systems (RDBMS). We can identify three important ones which are concerned with more schema, more declarative query languages and more data integrity to enhance data quality and business intelligence (BI) processing. In fact, excluding ACID (Atomicity, Consistency, Isolation and Durability properties expected from a RDBMS) and consistency issues [18], after these additions a NoSQL system may start resembling a RDBMS. We argue that the integration of these features needs to consider the semantics of the elements of the application domain. This could be a major break through for both NoSQL stores and the Semantic community since RDBMS is not really reactive in integrating semantics.

Ontology Based Data Access (henceforth OBDA) may be a good fit in this direction since it aims to represent the concepts and properties of a domain with a formalized ontology. OBDA provides a semantic conceptual schema over a repository of data and, due to its logical formalism, it is likely to support formal analysis, optimization and reasoning. In this paper, we focus on the currently most popular form of OBDA systems: those based on Description Logics (DL) [3]. DL-based OBDA is largely motivated by the Semantic Web and has mainly been studied for data repositories corresponding to RDBMS. The main contribution of this work is to show that OBDA is even more needed in the NoSQL ecosystem. Moreover, we consider that a common OBDA approach can be designed for both RDBMS and NoSQL, hence supporting a form of data integration from both these data management systems.

This paper is organized as follows. In Section 2, we present the background knowledge on OBDA and NoSQL. Section 3 introduces a social medical application that will serve as a running example. Section 4 tackles issues on the three features identified for NoSQL systems: schema modeling, declarative language and constraint violation detection. Section 5 contains a discussion and concludes the paper.

II. BACKGROUND KNOWLEDGE

In this section, we introduce the main notions needed to understand the concepts used in this paper. Basically, we

present the main characteristics of DLs and in particular the DLs that are used by OBDA in the context of the Semantic Web. Then, we present some of the most popular NoSQL data models, i.e. document and column family stores.

A. DL-based OBDA

DLs correspond to a fragment of first order logic with sound and complete inference procedures. They are generally used to represent the knowledge of a particular application domain and are composed of a TBox and an ABox that respectively specify the general properties and the instances of both concepts and roles. In this context, concepts and roles correspond respectively to unary and binary predicates.

In the context of OBDA, specific tractable description languages, denoted as the DL-Lite family [5]), have been defined to express conceptual data models (e.g. Entity-Relationship [7]) and object-oriented formalisms (e.g. basic UML class diagram[1]). Among this DL-Lite family, the so-called DL-Lite$_\mathcal{R}$ have been selected as the basis for the OWL2QL profile[2]. In this DL, the syntax of concept and role expressions is as follows:

$$\begin{array}{l} B \rightarrow A \mid \exists R \\ C \rightarrow B \mid \neg B \\ R \rightarrow P \mid P^{-} \\ Q \rightarrow R \mid \neg R \end{array}$$

where A denotes a concept, P denotes a role and P^{-} correspond to the inverse of the relation P.

TBox and ABox assertions are formed according to the following syntax (with a and b denoting constants):

$$\begin{array}{c} B \sqsubseteq C \text{ and } R \sqsubseteq Q \\ B(a) \text{ and } R(a,b) \end{array}$$

The DL-Lite family has been specifically designed with a perspective toward OBDA applications. For instance, MASTRO [4] enables the definition of constraints and provides reasoning services in the context of OBDA.

B. Document and Column family NoSQL stores

In the following, we present Document and Column Databases because they are richer than key-value pairs and also because many data structures (objects) can't be easily modeled as key-value pairs. Note that Graph Databases must be thought as Document Databases with a special document type with the additional quality of allowing to perform graph operations.

Document databases focus on storage and access optimized for documents as opposed to rows or records. The data model is collections of documents, which contain key-value collections. In a "document" the values can be nested documents or lists as well as scalar values. The attribute names are not predefined in a global schema but dynamically defined for each document at runtime. Moreover, unlike for a tuple, a wider range of values are authorized. A document store stores data in tree-like structures and requires the data to be stored in a format understood by the database. In theory, this storage format can be XML, JSON (JavaScript Object Notation), Binary JSON (BSON), or just about anything, as long as the database can understand the document internal structure.

MongoDB[3] is an open source, schema-free, document-oriented database using a collection oriented storage. Collections are analogous to tables in a relational database. Each collection contains documents that can be nested in complex hierarchies and still be easy to query and index. A document is a set of fields, each one being a key-value pair. A key is a string and the value associated can be a basic type, a document, or an array of values. In addition, it allows efficient storage of binary data including large objects (e.g. photos and videos).

MongoDB provides support for indexes and object queries for fetching data. Indexing techniques rely on B-Trees. Multi-key indexes are also supported. Dynamic queries are also supported with automatic use of indices, like RDBMSs. It has a query optimizer, and allows ad-hoc queries. MongoDB also supports map-reduce techniques for complex aggregations across documents. MongoDB provide access in many languages such as C, C++, C#, Ruby, Java, etc.

MongoDB scale reads by using replica sets and it scale writes by using sharding. It is tolerant of incomplete data. However it has less flexibility with querying (e.g. no JOINs between collections)

Column family databases or *big table*-like databases[6] are very similar on the surface to relational databases, but they are quite different because they are oriented differently to maximize disk performance. Here, the motivation is that generally, a query doesn't return every column of a record. They store their data such that it can be rapidly aggregated with less I/O activity. A *big table*-like database consists of multiple tables, each one containing a set of addressable rows. Each row consists of a set of values that are considered columns.

Cassandra[4] is a column family database having a data model that is dynamic and column-oriented. Unlike a relational database, there is no need to model all of the columns required by an application up front, as each row is not required to have the same set of columns and columns can be added with no application downtime. A table in Cassandra is a distributed multidimensional map indexed by a key. The value is an object that is highly structured and the row key in a table is a string. Columns are grouped together into sets called column families. Column Families contain

[1]http://www.omg.org/uml/

[2]W3C, "OWL2 Profiles", available at http://www.w3.org/TR/owl2-profiles

[3]http://www.mongodb.org

[4]http://www.cassandra.apache.org

multiple columns, each of which has a name, value, and a timestamp, and which are referenced by row keys. There are two kinds of column families: *Simple* and *Super*. Super column families stands for column family within a column family. The column families are fixed when a Cassandra database is created, but columns can be added to a family at any time.

The index of the row keys of a given column family serves as primary index. It is the responsability of each participating node to maintain this index for the subset of data it manages. Additionnaly, because each node is aware of ranges of keys managed by the others nodes, requesting rows can be more efficient. Cassandra supports secondary indexes, i.e. index on column values.

Cassandra allows fast lookups, and support for ordered range queries. Cassandra is recognized to be really fast for writes in a write-heavy environment. However, reads are slower than writes. This may be caused by not using B-trees and in-place updates on disk unlike all major relational databases and some NoSQL systems. In terms of data access, Cassandra has a very low level API that you access through its RPC serialization mechanism, e.g. *Thrift*. Recently, Cassandra query language(CQL) has appeared as an alternative to the existing API.

III. Running Example: Medical Social Application

In order to illustrate our approach, we present a medical social application which stores and processes patient information concerning their diseases, allergies, and drug prescriptions. We only propose an extract of the database and ontology that currently composes the real application.

Concerning the database, it is represented in Figure 1 as a set of JSON documents. It makes an intensive use of denormalization to support fast access to the data. Our data extract highlights three entities which we will denote as collections, namely `Patient`, `Disease` and `Drug`, which concretely illustrate the kind of reasoning and query rewriting one can perform with OBDA. The `Patient` collection stores information on the patient (e.g. last and first names, gender, date of birth, etc.), the kinds of allergies and diseases she is suffering from and the list of treatments she is following. Information related to the treatments contain the start date and the (optional) end date, the drug identification and name. Several interesting features of this application require to reason over drugs, molecules and diseases data and knowledge. The `Disease` collection stores information on a particular disease. The `Drug` collection regroups information such as name, molecule name, posology, etc. on a drug product. They both contain a list of patients involved.

One important aspect in this social application is the ability to integrate existing ontologies. For instance, the Linked Open Data[5] proposes an access to the Diseasome ontology and the DBPedia repository proposes access to the Anatomical Therapeutic Chemical system (ATC)[6] that classifies drug molecules. Thus in many application domain, the integration of an ontology comes for free since it is possible to reuse high quality and updated ontologies in a format compatible with the Semantic Web.

Table I
Ontology of the social medical application

1.	$Patient \sqsubseteq Person$	5.	$\exists sufferFrom^{-} \sqsubseteq Disease$
2.	$Patient \sqsubseteq \exists lastName$	6.	$Patient \sqsubseteq \exists sufferFrom$
3.	$\exists lastName- \sqsubseteq String$	7.	$Disease \sqsubseteq \exists sufferFrom^{-}$
4.	$\exists sufferFrom \sqsubseteq Patient$		

Based on this database instance, we now propose in Table I an associated DL-Lite$_{\mathcal{R}}$ ontology.

The axioms of this ontology extract state that a patient is a person (1), a patient has a last name (2) which corresponds to a string of characters (3), similar axioms can be stated for gender, first name, birth date, etc. Axioms (4-7) specify that the domain and range of the `sufferFrom` role are respectively the `Patient` and `Disease` concepts. A similar pattern can be defined to state that a patient follows a treatment and that a treatment contains a drug which cures a disease.

IV. Features of an OBDA system for NoSQL stores

In this section, we tackle the issue of supporting three important features desired in NoSQL stores: adding schema, providing a declarative query language and supporting integrity constraints. In Figure 2, we provide an overview of an architecture composed of three layers: query, semantic and storage. The Storage layer is composed of standard NoSQL databases but in this paper we concentrate on a single instance (look for [9] for more details on integrating several instances). The Semantic layer is the cornerstone of this research and is dealt with in sub-sections IV-A and IV-C. The Query layer is treated in sub-section IV-B In this architecture, an end-user writes a SPARQL query[7] which is sent to the OBDA system. There it is translated using mapping assertions and inferences over the ontology into a set of queries that are executed on the NoSQL sources.

A. Schema features

Many functionalities depend on the addition of a schema to NoSQL data models. Some of them are discussed in the next sections of this work: generation and optimization of queries, detection of integrity constraints violation.

[5] http://linkeddata.org

[6] http://www.whocc.no/atc_ddd_index/

[7] http://www.w3.org/TR/rdf-sparql-query/

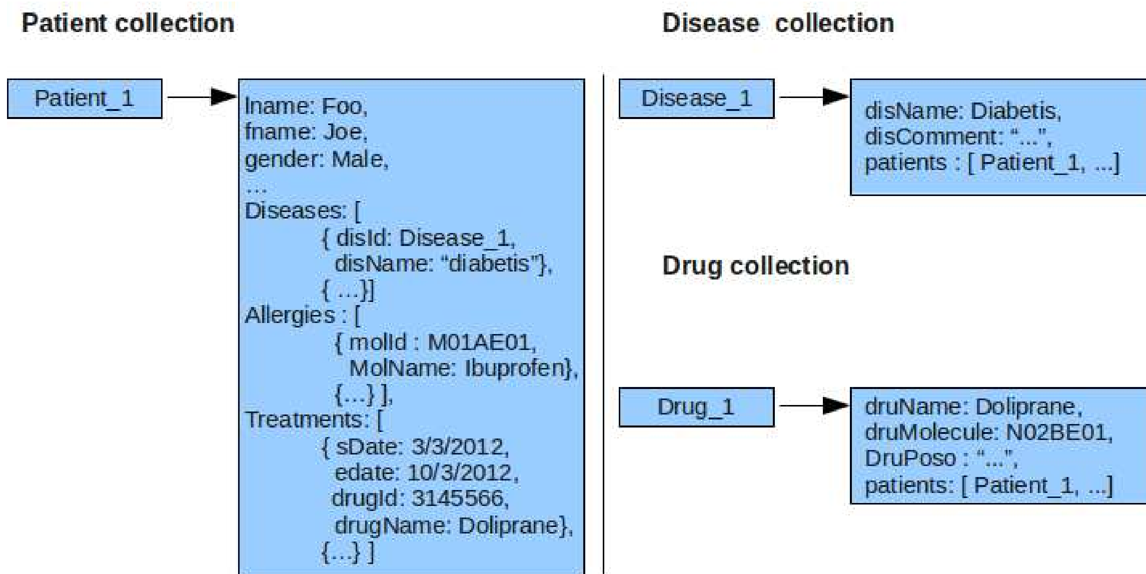

Figure 1. Database of the medical social application

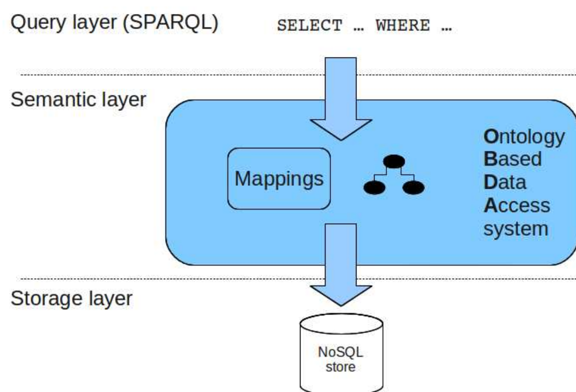

Figure 2. Overview of OBDA architecture

Integrating a schema may be considered a complex task as long as we want to retain the flexibility of schema-less stores. Recall that this flexibility supports the efficient storage of sparse data and eases the insertion of additional fields in tuples. But tackling this addition is nevertheless a big opportunity to integrate functionalities that we do not get from existing database management systems, e.g. dealing with the semantics of the underlying application domain. We consider that it is the right timing for proposing such an integration due to the recent emergence of NoSQL, its receptiveness, reactivity and active community. It is known that incorporating a novel feature is harder in well-established ecosystems. For instance, the DL community started several years ago to motivate and propose OBDA as an alternative for conceptual schemata to the RDBMS market. But these efforts have not payed off yet and all major RDBMS vendors remain closed to the principle of OBDA.

The difficulty of maintaining flexibility and integrating a schema is related to (1) the notion of mapping schema and instance elements and (2) query answering in the context of a set of mapping assertions. Mapping assertions enables to define correspondences between the target elements (i.e. ontology concepts and roles in this work) and the sources (i.e. keys and collections of NoSQL stores). Once these correspondences are specified, queries can be expressed in a query language of the target, translated in constructs accepted by the sources which are executed to retrieve results.

Concerning the mapping solution, it also needs to emphasize a form of flexibility by not imposing one-to-one correspondences between schema and instance elements. Thus an instance element (e.g. a key) may not be mapped to an ontology element and some schema elements may not be mapped to an instance element. This yields an approach where the design of the ontology and the mapping assertions correspond to the specific needs of an application. For instance, one may design a version of our medical application where allergy related information is not considered but knowledge on diseases and drug molecules are stored in the ontology. Note that this approach fits the best practices of application developers using NoSQL stores. That is they tune the structure of the database instances to fit the peculiar needs of an application, ensuring that certain queries will perform very efficiently while many others may be impractical.

In [9], we have proposed a mapping solution between a relational schema and a set of NoSQL stores/RDBMS.

This mapping language can easily be adapted to the domain of ontologies. It thus enables to link a single ontology to a set of NoSQL sources. A main contribution of this work was to integrate the notion of an access path in the mapping assertions. This enables to tackle the issue of the denormalized aspect of NoSQL stores. In fact, it enables to reply to the following question: how one can relate an ontology element to a NoSQL element that can be found in several entities? For instance in our running example, this is the case with patient identifiers which can be found in the Patient, Disease and Drug collections. Depending on the query asked, our system selects the mapping assertion with the most efficient access path. This will support the generation of a query that will retrieve information with optimized performance.

We now adapt the mapping language of [9] to a target ontology. It corresponds to the GAV (Global As View) approach with sound sources [12] which stores possible access paths for a target element. To specify access paths, an end-user either specifies the '*' symbol which, like in SQL, denotes the complete list of attributes of a given collection or an attribute name, denoting that the source entity offers an efficient access, either using a key or an index.

Definition 1 General syntax of a mapping assertion with an access path specification on attribute α of the source is as follows: EntityO $\overleftarrow{\alpha}$ EntityS($< key; value >$) where entityO and EntityS respectively denote a conjunction of ontology elements and a conjunction of collections or column families. Due to the schema flexibility of NoSQL databases, we can not rely upon any attribute ordering in a collection or column family. Hence, we must use attribute names to identify distinct portions of a tuple. In order to map EntityO and EntityS attributes, we introduce a 'AS' keyword to define a correspondence between attribute symbols of the mapping assertion. An entry of EntityO takes the form of either a concept assertion($C(a)$ with C a concept and a an individual) or a role assertion ($R(a,b)$ with R a role and a,b individuals). The individual labels used in EntityO must correspond to the ones used in EntityS which we are now specifing. An entry in EntityS is defined as a key/value structure using a '<key ; value>' syntax, where key is either (i) 'PKEY AS k' or (ii) a variable name (previously defined in a EntityS couple of the same mapping) and value is either of the form (i) nameS AS nameO (where nameS and nameO are resp. attribute names from the source and the target) or (ii) of the form of a possibly nested list comprehensions $[item|item \leftarrow list]$ (where item corresponds to an element of the set denoted by list which is an attribute identifier of the source). Finally, a keyword is introduced to denote the primary key of the structure (i.e. 'PKEY AS') and to manipulate it, e.g. IN KEY.

Example 1 We now present two mapping assertions in the context of our running example:

1	$Disease(k)$, $dName(k,n)$, $dLabel(k,c)$, $Patient(p)$ $sufferFrom(p,k)$	$\overleftarrow{k,n,c}$	Disease(<PKEY AS k; disName AS n, disComment AS c, [p | p←Patients]>)
2	$Patient(k)$, $lastName(k,n)$, $firstname(k,f)$, $gender(k,g)$ $Disease(d)$, $sufferFrom(k,d)$	$\overleftarrow{k,n,f,g}$	Patient(<PKEY AS k; lName AS n fName AS f, gender AS g, [d | d ← Diseases]>)

In these two mapping assertions recall that elements on the left (resp. right) hand side of the arrow correspond to ontology (resp. NoSQL) elements.

Mapping #1 enables to retrieve information related to a disease. Each tuple in the NoSQL store will generate an individual in the KB with type $Disease$. This individual will have $dName$ and $dLabel$ properties storing respectively the name and a comment on this disease. Finally, a patient individual will be created in the KB for each patient known to suffer from this disease. Note that the access path specified for this mapping are the aliases k, n and c corresponding to a disease identifier, name and comment. That is this mapping is an efficient access path if one wants to retrieve information from these attributes. In the case of an absence of secondary index on the patients field of the Disease collection, it is certainly not efficient to retrieve all Patient or sufferFrom information from this mapping assertion since it would require a complete scan of all disease entries. Nevertheless, for retrieving all patients suffering from a given disease, this mapping is a good option.

In the next section, we explain how access path selection impacts the generation of queries.

B. Declarative Query Language

A main advantage in using a DL-based OBDA approach is to enjoy all the stack of technologies developed and maintained in the Semantic Web. Among them, RDF (Resource Description Framework)[8] plays a central role. It corresponds to a directed, labeled graph data composed of so-called triples, i.e. subject, predicate and object. Considered as a data model, RDF comes with a query language named SPARQL. It consists of a set of patterns which are matched against an RDF graph. Elements of a pattern can be a URL, a variable (starting with a '?' symbol) or a literal (only for objects). Hence, a SPARQL query can be represented as a graph. A frequently encountered query pattern takes the form of a star since there is a central node in the query from which several edges are departing. The identification of a star query can be computed from measures, e.g. degree centrality, of the centrality of vertices within a graph. Intuitively, the node with the greatest number of links is considered central.

[8]http://www.w3.org/RDF/

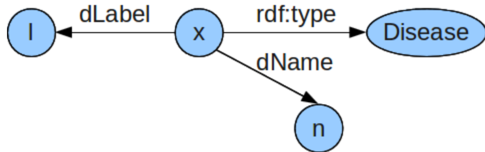

Figure 3. Graph representation of Example 1's query

Example 2 The following query retrieves the name and comment of all stored diseases:

```
SELECT ?n,?c WHERE {?x rdf:type
Disease.?x dName ?n. ?x dLabel ?c}.
```

From the graph representation of Example 2's query displayed in Figure 3, it is clear that the variable 'x' is the central node of this star-query.

As shown in Figure 2, SPARQL queries sent to the OBDA system are translated into queries compatible with the NoSQL store. This kind of query answering is referenced as on-demand since the information stays at the sources and is retrieved from queries expressed over the target schema. This requires a translation since the query facilities available at both the target and the source are different, i.e. SPARQL for the ontology and a specialized query language (e.g. MongoDB) or some procedural code using an API (e.g. CouchDB[9]). This translation implies two operations: (1) generating several queries using ontology-based inferences and (2) searching an efficient rewriting using mapping assertions and their access paths.

Consider a SPARQL query asking for properties of an individual of type A and an ontology containing axioms $B \sqsubseteq A$, $C \sqsubseteq A$ and $B \sqsubseteq \neg C$. The KB may not contain individuals explicitly defined as instances of A. This does not mean that the result set of this query ought to be empty since the KB may contain individuals of types B and C. Hence, it is necessary to generate queries based on computing the transitive closure of the concept A. A standard DL reasoner or even explicitly stated subsumption hierarchies enables to generate queries in an efficient and transparent manner.

Example 3 In our running example, ATC codes are explicitly represented as a hierarchy of DL concepts. With our inference enabled approach, it is sufficient to express the following query to retrieve all twelve Opium alkaloids and derivatives that act as cough suppressants.

```
SELECT ?x WHERE {?x rdf:type Molecule.
FILTER regex(?x,``R05DA'').}
```

This is due to the concept hierarchy stating that $X \sqsubseteq R05DA$ with X $\in \{R05DA01, .., R05DA12\}$.

Concerning the generation of efficient rewriting of target queries over the sources, our approach mainly uses the set of mapping assertions and their access paths. The main principle is to detect the central node of a (star-) query and to search for a mapping assertion whose access path is defined over the type of this central node.

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

Example 4 In the case of the query of Example 2, the mapping assertion selected for the translation is the first of Example 1. This is motivated by the type of `?x` being `Disease` and the access path being the most performant for this mapping.

We have already emphasized that the set of queries executed in an application using a NoSQL store is well defined and motivates the manner one denormalizes the database. This adequacy between application queries and physical storage of the data is transposed into the mapping assertions. This enables to define queries that may not require joins although accessing information related to different domain entities.

Example 5 Consider that our social medical application enables an end-user to search for all male patients and their diseases. This would correspond to the following query:

```
SELECT ?n ?f ?d WHERE {?p rdf:type
Patient. ?p lastName ?n. ?p firstName
?f. ?p gender ?g. ?p sufferFrom
?d. ?d rdf:type Disease. FILTER
regex(?g,"male")}
```

This is clearly a star-query with `?p` being the central node. Although the information returned by the query correspond to the last and first names of a patient as well as disease name, no join is needed. Thus, mapping assertion #2 is clearly the most efficient.

Finally, it may be impossible to identify a single central node in a query. In classical denormalized schemata, this is encountered in queries requiring joins. Note that for applications requiring an intensive use of such queries, a denormalization is certainly preferred solution. For instance with Example 6's query, one may model the database instance with both disease name and comment in the `Patient` collection, hence replying without a join. Nevertheless, using heuristics, a translation may be possible be without guaranteeing high performance query answering. Due to paper size restrictions, we only provide a sketch of a strategy we have designed. Intuitively it consists in partitioning the query into a set of star-queries and taking advantage of roles relating them, denoted linking-roles. For instance the original query can be partitioned into `n` star-queries if `n` central nodes are identified. Then each of them can be recursively partitioned into star-queries until all sub queries can not decomposed. These atomic queries can be translated using the approach previously explained and each of them are being joined from operations performed on linking-roles.

Example 6 Consider the following query that retrieves the label and name of diseases as well as the last and first names of patient who are suffering from this disease. Its graphical representation (Figure 4) clearly emphasizes that it is not a star-query since both nodes `p` and `x` have the same centrality degree. Hence the query can be partitioned into two atomic

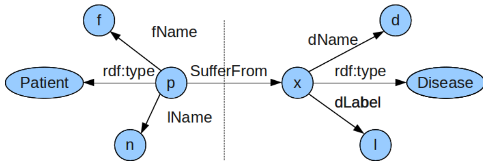

Figure 4. A non star-query

star-queries: the ones on each sides of the dashed line. Each of them will be related using the `sufferFrom` role.

Our translation mechanism generates queries taking the form of peculiar NoSQL query language or of a program using a API provided by the NoSQL system. Currently, our prototype generates Java programs for MongoDB and Cassandra but we are working on generating CQL queries for the latter. The heterogeneity of query language offers among NoSQL systems makes it difficult to design an interoperable framework. The ability to transform any NoSQL model into a graph confirms that SPARQL is a good candidate for filling the role of a common query language.

C. *Integrity Constraints*

In this paper, we consider that ontologies are correct or are debugged using tools such as [10] or [16]. This implies that inconsistencies must come from the data. This problem recently started to draw some attention and two solutions can be identified: (1) dealing with inconsistency-tolerant query answering and (2) providing data quality tools.

Inconsistency-tolerant query answering has been identified to be intractable in data complexity for DL-Lite [11] and the EL family [15]. It is the subject of on-going research to develop algorithms and heuristics to practically solve this problem in an efficient manner.

In this paper, we focus more on the second approach which aims to detect the insertion of inconsistent data. The mechanisms found in RDBMS is a major inspiration in dealing with this issue: checking the violation of integrity constraints, i.e. logical axioms such as functional or inclusion dependencies [1], whenever a tuple is updated. This main mechanisms associated to this detection are prevention, i.e. forbidding the persistent storage of erroneous data, or correction, i.e. cleaning the data to restore a correct value [2]. Some of the axioms stored in the ontology can serve as integrity constraints but one has to be careful with the semantics adopted by DL [14]. The main difference in terms of semantics between a KB and an RDBMS is the Closed/Open World Assumption (henceforth CWA and OWA). Intuitively, in OWA, one cannot assume that the knowledge in the KB is complete while it is in CWA.

Several solutions have been identified to support integrity constraints with an ontology [17]. A simple approach consists in finding relevant axioms that could act as integrity constraints and representing them as queries which are executed when a tuple is updated. Finding the axioms may be performed automatically or more or less involving an end-user. An automatic discovery is generally based on a pattern matching approach, e.g. discovering all axioms matching an inclusion dependency, but its results is usually less relevant than a manual one. The manual discovery involves more work from the end-user but has much more fine-grained results. Some semi-automatic methods certainly have to be designed and implemented. In [8], a non-standard set of integrity constraints is identified from an ontology (e.g. functional and inclusion dependencies with conditions) and associated operations to detect violations are described with optimized algorithms.

Example 7 Consider that the `Drug` collection of our running example stores information on the social security reimbursement rate and the type of a product (i.e. homeopathy, allopathy, etc.). A logical axiom stating that only allopathy drugs can be reimbursed with a rate of 65% is expressed as:
drug (DRUGRATE=65 $\rightarrow$ DRUGTYPE='allopathy') and corresponds to a conditional functional dependency since it depends on the constant values '65' and 'allopathy'.

Once these axioms are identified, it is generally simple to translate them into a compliant query language: SPARQL. The generation of such queries is straightforward and can be automatized.

Example 8 The query generated from the integrity constraints of Example 7 is as follows:

```
SELECT ?drug WHERE { ?drug rdf:type :Drug.
 ?drug :drugRate ?rate. FILTER (?rate=65).
 ?drug :drugType ?type. FILTER
 (?type != "allopathy").}
```

The next step is to execute a set of SPARQL queries when a tuple is updated. Like in a RDBMS, SPARQL queries are triggered from tuple updates. A naive approach of this step is quite inefficient since the set of SPARQL queries representing integrity constraints may be very large for certain domain of interest. Hence, it is important to identify a subset of these queries that are relevant to update tuple. This operation is performed using the mapping assertions which relate NoSQL elements to concepts and roles of the ontology. For instance, in the context of a drug tuple update, it is not relevant to check integrity constraints related to patients, allergies and diseases.

Finally, it is important to provide a high performance method to detect data inconsistencies. After all, NoSQL stores have been designed and implemented for high throughput and speed. We argue that a tunable approach to triggering SPARQL queries is needed. Since developers using NoSQL stores generally control sensitive aspects of their applications, e.g. handling consistency level, it may be preferable to supply them with control of integrity constraints checking. The levels we are proposing range from:

- a fine-grained tuple update checking where each time

a tuple is updated, a set of queries is triggered.

- a time dependent level, i.e. the developer programs at which period of time checking is to be performed.
- an adaptable runtime dependent level where depending on the workload (IO operations) the system may decide to perform live or delayed checking.

The first level may slow the system down but ensures that at all states, the database instance is consistent. On the opposite, the last two levels may not impact the system performance but allow inconsistencies during a certain period of time. Something that may not be that annoying in domains using NoSQL stores, i.e. eventually consistency.

V. Discussion

This paper motivates the fact that OBDA is able to fulfill some of the most desired features in NoSQL databases. The features considered in this paper are adding schema facilities, providing a declarative query language and supporting integrity constraints. For all three of them, we have presented some proofs of concepts but could not, due to space limitations, provide more details.

Nevertheless, we have shown that using OBDA provides functionalities that go beyond the expectation of most NoSQL developers and users, e.g. reasoning over semantic schemata. Moreover, adopting an OBDA approach may not come at an extract cost for end-users. In many cases, they would not have to design from scratch the ontology of their application domain. Thanks to available repositories, one may be able efficiently find, via tools like Watson or Liked Open Data, and reuse existing ontology and knowledge base.

We consider that much work is needed in the direction of integrating OBDA with NoSQL stores. In [13], the authors highlight the duality of RDBMS and NoSQL (which they call CoSQL) and argue that the NoSQL follows OWA. We can consider that the flexibility of NoSQL is related to this open world semantics. Precisely, they oppose the Closed World Assumption of RDBMS to the Open World Assumption (OWA) of NoSQL which is at the source of its flexibility. Moreover, ontologies and Knowledge Bases (KB) are also evolving in an OWA context, proving their adequacy with NoSQL stores. For instance, the impact of having an open world assumption on both the ontology and the NoSQL system is worth studying. With a more practical approach, the impact of new constructs in the yet to be released SPARQL 2 or the integration of SPARQL queries within the MapReduce data processing paradigm needs to be investigated. Since some NoSQL stores correspond to distributed hash tables, it is certainly preponderant to do research on the distribution and parallel processing of the features presented in this paper.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79, 1999.

[3] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[4] D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, R. Rosati, M. Ruzzi, and D. F. Savo. The mastro system for ontology-based data access. *Semantic Web*, 2(1):43–53, 2011.

[5] D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *dl-lite* family. *J. Autom. Reasoning*, 39(3):385–429, 2007.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In *OSDI*, pages 205–218, 2006.

[7] P. P.-S. Chen. The entity-relationship modeltoward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, Mar. 1976.

[8] O. Curé. Improving the data quality of drug databases using conditional dependencies and ontologies. *ACM Journal of Data and Information Quality*, to appear, 2012.

[9] O. Curé, R. Hecht, C. L. Duc, and M. Lamolle. Data integration over nosql stores using access path based mappings. In *DEXA (1)*, pages 481–495, 2011.

[10] A. Kalyanpur, B. Parsia, E. Sirin, and J. A. Hendler. Debugging unsatisfiable classes in owl ontologies. *J. Web Sem.*, 3(4):268–293, 2005.

[11] D. Lembo, M. Lenzerini, R. Rosati, M. Ruzzi, and D. F. Savo. Inconsistency-tolerant semantics for description logics. In *RR*, pages 103–117, 2010.

[12] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[13] E. Meijer and G. M. Bierman. A co-relational model of data for large shared data banks. *Commun. ACM*, 54(4):49–58, 2011.

[14] B. Motik, I. Horrocks, and U. Sattler. Bridging the gap between owl and relational databases. In *WWW*, pages 807–816, 2007.

[15] R. Rosati. On the complexity of dealing with inconsistency in description logic ontologies. In *IJCAI*, pages 1057–1062, 2011.

[16] S. Schlobach, Z. Huang, R. Cornet, and F. van Harmelen. Debugging incoherent terminologies. *J. Autom. Reasoning*, 39(3):317–349, 2007.

[17] J. Tao, E. Sirin, J. Bao, and D. L. McGuinness. Integrity constraints in owl. In *AAAI*, 2010.

[18] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.