Extending Ontology-Based Databases with Behavioral Semantics

Youness Bazhar[1], Chedlia Chakroun[1], Yamine Aït-Ameur[2], Ladjel Bellatreche[1], and Stéphane Jean[1]

[1] LIAS - ISAE ENSMA and University of Poitiers,
Futuroscope, France.
{bazhary, chakrouc, bellatreche, jean}@ensma.fr
[2] IRIT - INPT ENSEEIHT,
Toulouse, France.
yamine@enseeiht.fr

Abstract. Recently, ontology-based databases (OBDB) have been developed as a solution to store and manipulate, efficiently and in a scalable way, ontologies together with data they describe. Currently, existing OBDB propose weak solutions to calculate derived (non-canonical) concepts. Indeed, these solutions are internal to the OBDB and specific to the ontology model (formalism) supported. As a consequence, non-canonical concepts can not be in all cases properly defined with the different available mechanisms since existing solutions are not constantly suitable. In this paper, we propose a generic solution which is an extension of OBDB with the capability to introduce dynamically operators to calculate non-canonical concepts. These operators can be implemented in different ways (e.g. with external programs or with web services). Then, we show the interest of this extension by improving a proposed methodology to design databases storing ontologies. Finally, a prototype implementing our design approach is outlined.

1 Introduction

Ontologies are used to express the semantics of a domain through conceptual, formal and consensual models. Nowadays, they are used in many domains such as engineering, e-commerce or chemistry, and they have contributed to the success of many applications such as systems integration, information retrieval or natural language processing. Several languages and formalisms exist to define ontologies. They are often dedicated to a specific domain such as the semantic web (e.g., OWL [13], RDF [19] and RDF-Schema [6]) or engineering (e.g., PLIB [24, 25]).

All ontologies are not alike. Three types are often distinguished: (1) *canonical ontologies* that include only primitive concepts, (2) *non-canonical ontologies* that extend canonical ontologies with derived concepts i.e., notions expressed in term of other concepts and (3) *linguistic ontologies* that associate a linguistic definition to canonical and non-canonical concepts.

With the growing size of data described by ontologies, the need to efficiently persist data together with ontologies, in a scalable way, appeared. As a consequence, a new type of databases has been proposed, called ontology-based databases (OBDB), that stores ontologies and data in the same repository. Several architectures of OBDB have emerged in the last decade. They differ in (i) the way they support ontology model(s), (ii) the separation of the different data layers and (iii) the database schema they use to store ontologies and data.

Currently, existing OBDBs support mainly canonical ontologies. On the contrary, non-canonical concepts are often managed with frozen and hard-encoded operators offered by the OBDB or by reasoners like Racer[3] or Corese[4] in central memory. In databases, mechanisms such as views, triggers or stored procedures are set up to compute derived classes and properties. Consequently, all the available mechanisms used to compute derived concepts are internal to the OBDB become specific to the supported ontology model.

The aim of the work presented in this paper is to provide a generic solution to handle non-canonical concepts in OBDB. Our idea is that non-canonical concepts can be both computed by internal mechanisms of databases as well as external mechanisms such as reasoners, web services and remote programs. Our approach borrows concepts and technique from persistent meta-modeling systems (PMMS) [2]. It consists in extending PMMS to define OBDBs with the capability to introduce dynamically operators that can compute non-canonical concepts independently of any ontology-model. These operators can invoke methods of reasoners, external programs, web services or use internal mechanism of the database. Thus, this extension gives to OBDBs an additional power of expressiveness and a better support of non-canonical ontologies. To show the interest of our approach, we propose an OBDB design methodology which handles non-canonical concepts by invoking external reasoners before building the logical model of the database.

The remainder of this paper is organized as follows. Section 2 presents the OntoDB ontology-based database, on which our approach is based, together with its associated exploitation language, OntoQL. An example illustrates the drawbacks of this system. Section 3 exposes different OBDB architecture types we have identified and the way they support non-canonical concepts. Section 4 introduces the extension of OBDBs with the flexible support of non-canonical concepts. The proposed OBDB design methodology which supports non-canonical concepts using ad hoc mechanisms to compute non-canonical concepts is given in section 5. This section shows an application of our approach to improve the OBDB design methodology. Finally, section 6 gives a conclusion and some future research directions.

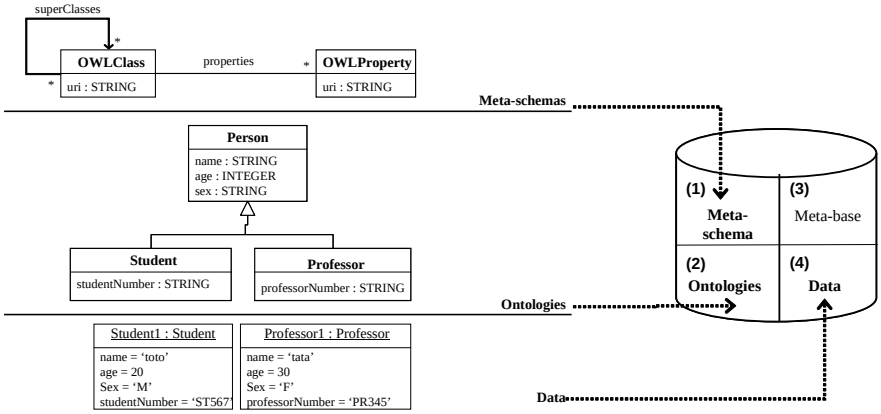

Fig. 1. The OntoDB architecture

2 OntoDB: an ontology-based database

OntoDB [14] is an ontology-based database architecture with four parts (Figure 1). The *meta-base* (3) and *data* (4) parts are the classical parts of traditional databases. The meta-base part contains tables used to manipulate the whole data of the database, and the data part stores data described by ontologies. The other two additional parts, i.e. *meta-schema* (1) and *ontologies* (2) parts store respectively ontology models (formalized by a meta-model) and the ontologies as instances of ontology models.

OntoDB stores data in relational tables, it is implemented on the PostgreSQL RDBMS. Figure 2 shows an example of data layers representation in OntoDB. This example defines a simple ontology model with two entities: `OWLClass` and `OWLProperty`. Each entity is associated to a corresponding table at the ontology level. Similarly, each class of an ontology is associated to a table at the data level to store classes instances.

Once data of the different abstraction levels are stored in OntoDB, we observe that the traditional database exploitation languages like SQL are not powerful enough. These languages require a deep knowledge of the database representation used by the OBDB for the different layers of data. In this context, as a next step, the OntoQL language [16], [17] has been proposed to manipulate ontology models, ontologies and data without any knowledge of the database representation used for storing all data. Next subsections summarizes OntoDB/OntoQL capabilities.

[3] http://www.racer-systems.com/
[4] http://www-sop.inria.fr/edelweiss/software/corese/

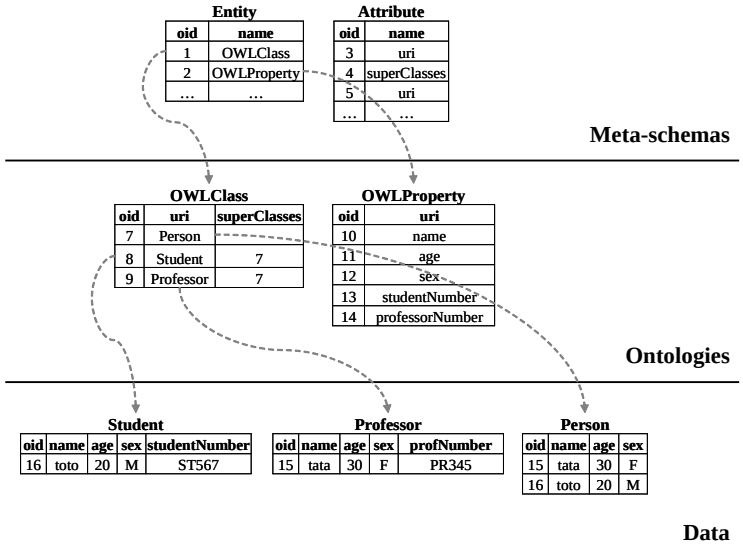

Fig. 2. Representing data of the different abstraction layers in OntoDB

2.1 Meta-schema evolution

The meta-schema part of OntoDB can be enriched to support new ontology models using the OntoQL language. For instance, the meta-schema of our example can be created with the following statements.

```
CREATE ENTITY #OWLProperty (#uri STRING);
CREATE ENTITY #OWLClass (#uri STRING, #superClasses REF(#Class) ARRAY, #properties
REF(#OWLProperty) ARRAY);
```

These two statements create a simple meta-schema containing two entities: `OWLProperty` and `OWLClass` with different attributes. Once the meta-schema part is enriched, OntoDB supports the creation and the manipulation of ontologies conforming to the defined ontology model (see Figure 1 and 2).

2.2 Ontologies definition

OntoQL possesses also the capability to create ontologies that conform to the supported ontology models. Next statements create the ontology of the example of Figure 1 and 2.

```
CREATE #OWLClass Person (name STRING, age INT, sex STRING);
CREATE #OWLClass Student UNDER Person (stdtNumber INT);
CREATE #OWLClass Professor UNDER Person (ProfNumber INT);
```

These statements create three classes (`Person`, `Student` and `Professor`) with different properties. Note that the keyword `UNDER` express inheritance relationships between classes.

2.3 Instances definition.

Once ontologies are created with OntoQL and stored in OntoDB, they can be instantiated to create classes individuals with a syntax similar to SQL. Next statements create individuals of our example (Figure 1 and 2).

```
INSERT INTO Professor VALUES ('tata', 'F', 30, 'PR345');
INSERT INTO Student VALUES ('toto', 'M', 20, 'ST567');
```

These two statements create one instance of each of `Professor` and `Student` classes.

2.4 Limitations of OntoDB/OntoQL to support non-canonical concepts

Currently, OntoDB does not use a specific mechanism for supporting non-canonical concepts. Let us consider, for example, the definition of a class as the union of other classes. For instance, a class `SchoolMember` could be defined from the union of `Professor` and `Student` classes. A possible semantics of the union of classes induces that the `SchoolMember` class becomes a super class of `Professor` and `Student` classes and conversely, `Professor` and `Student` become subclasses of `SchoolMember`. Besides, instances of `SchoolMember` class are obtained with the union of instances of `Professor` and `Student`. Here appears the need of computing a new class and its instances (non-canonical) from existing classes and instances (canonical). Such a construction is not available in OntoQL.

Our objective is to extend this language with the support of non-canonical concepts. Our idea is to be able to create the `SchoolMember` class with a statement that looks like:

```
CREATE #OWLClass SchoolMember
AS unionOf (Professor, Student)
```

Moreover, our aim is to provide a generic and a flexible solution that use internal mechanisms of databases as well as external mechanisms (reasoners, web services, etc.).

3 Related work: The support of non-canonical concepts in OBDB

To analyze the support of non-canonical concepts by OBDB, we use a classification of OBDBs. Indeed, different OBDB architectures are available. They store ontologies together with data they describe. We distinguish three types of OBDB architectures:

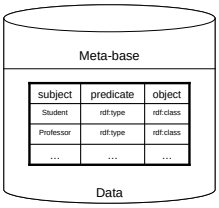

Fig. 3. Type1 OBDB

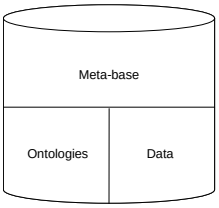

Fig. 4. Type2 OBDB

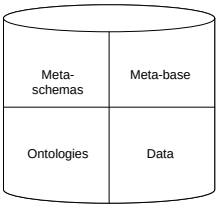

Fig. 5. Type3 OBDB

- **Type1 OBDB** (Figure 3) store both ontologies and data in a single triple table composed of `(subject, predicate, object)` that represent respectively the subject, the predicate and the object of triples. This OBDB architecture is used to store RDF-based ontologies. Some examples of type1 OBDB are 3Store [15], Jena [9], Oracle [11] and Sward [23]. To process non-canonical concepts, some of these OBDB provide hard encoded operators using deductive rules [11] and other OBDB use external reasoners [15].
- **Type2 OBDB** (Figure 4) store ontologies and data in two separated parts (ontologies and data parts). Main type2 OBDB are RDF Suite[1], Sesame [7], RStar [18], DLDB [21] and OntoMS [22]. These OBDB are mainly dedicated to store ontologies expressed with a specific model (e.g. RDFS, OWL, PLIB, etc.). Here different ontology models can be statically set up but they are frozen and cannot be modified nor extended. Type2 OBDB use different mechanisms to manage non-canonical concepts. Indeed, some OBDB use views [21], labeling schemas [22] or subtable relationships of object-relational databases [1, 7]. Other OBDB compute non-canonical concepts using mechanisms of deductive databases, typically logic-based engines like Datalog engine or reasoners [20, 28, 5, 21].
- **Type3 OBDB** (Figure 5) add another abstraction layer to type2 OBDB architecture. The added part aims at supporting different ontology models and their evolution. Thus, several ontology models (formalisms) can be dynamically supported with type3 architecture. The main example of type3 OBDB is OntoDB [14] which we presented in the previous section.

Synthesis. We notice that the different OBDB architectures focus on the way they store data and ontologies. However, non-canonical concepts are not powerfully addressed. Indeed, existing OBDB use some mechanisms that are either specific to the OBDB or to the supported ontology model. All the proposed solutions define static and hard encoded operations. No flexible solution to handle non-canonical concepts is available to support the definition on the fly at runtime of non-canonical operations.

4 Handling non-canonical concepts in ontology-based databases

4.1 A formal model

- $\mathbb{DT} = \{dt_1, dt_2, ..., dt_n\}$ is the set of simple or complex data types supported by OntoDB/OntoQL.
- $\mathbb{OP} = \{op_1, op_2, ..., op_n\}$ is the set of operations defined in OntoDB.
- $\mathbb{IMP} = \{imp_1, imp_2, ..., imp_n\}$ is the set of descriptions of implementations that may be associated to the defined operations.
- Each operation can be associated to a set of implementations.
- *implem* is a function that associates implementations to a given operation.
$implem : \mathbb{OP} \rightarrow P(\mathbb{IMP})$.
$\forall op_i \in \mathbb{OP}$, $implem(op_i) = Imp$, where: $Imp \subseteq \mathbb{IMP}$.
- *input* is a function that returns the input data types of an operation:
$input : \mathbb{OP} \rightarrow P(\mathbb{DT})$, where: $\forall op_i \in \mathbb{OP}, input(op_i) = dt$, where: $dt \subseteq \mathbb{DT}$.
- *output* is a function that returns output data types of an operation:
$output : \mathbb{OP} \rightarrow P(\mathbb{DT})$, where: $\forall op_i \in \mathbb{OP}, output(op_i) = dt$, where: $dt \subseteq \mathbb{DT}$.

The class diagram of Figure 6 illustrates these definitions.

4.2 Implementation

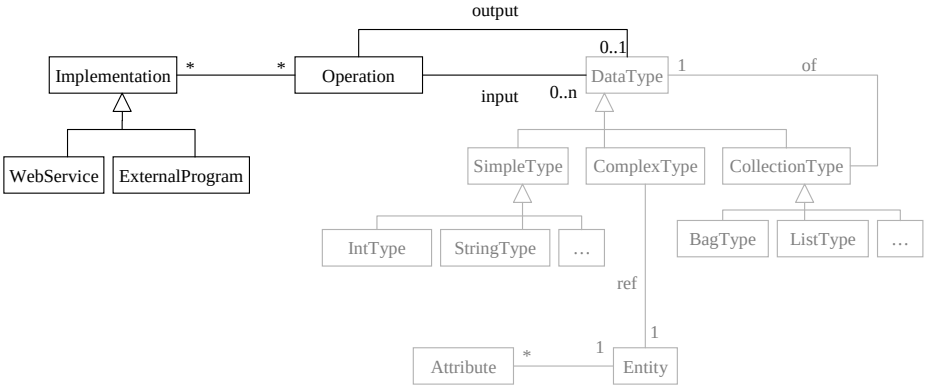

Fig. 6. The extension of the OntoDB/OntoQL meta-model with operations

The meta-model of OntoDB supports the gray part of the data model of Figure 6. Supporting operational behaviors (that encode non-canonical computations) requires an extension of OntoDB meta-model. So, the meta-model supported by OntoDB/OntoQL architecture is extended with new entities (Figure 6, black part).

– *Operation*: stores informations about operations like the operation name, its input and output data types and its implementation. Each operation is associated to an implementation that can be either a remote service or an external program or any other possible implementation. Information stored in the operation entity can be considered as operation specifications.
– *Implementation*: records implementations details. For example, if the operation implementation is a web service, the implementation entity stores the url, the namespace and the operation name, and if it is an external program, the implementation entity stores the location of the program, the package name, the class name, the method to invoke, etc.

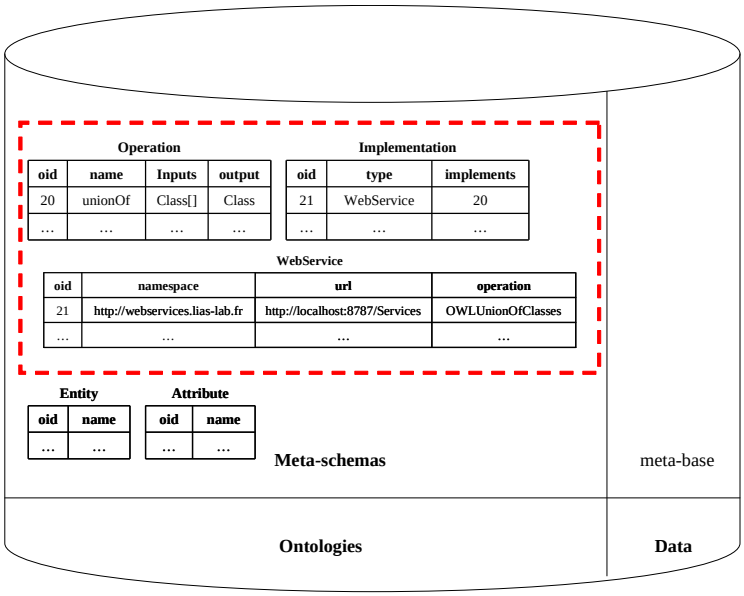

Fig. 7. The OntoDB extension

Once this extension is set up, the meta-schema part of OntoDB contains new tables to support operations (Figure 7). In this example, the operation table contains the specification of `unionOf` operation which processes the union of classes. The implementation table contains implementations associated to the defined operations. In our example, unionOf is implemented with a web service.

Once the OntoDB architecture is extended, we have also to extend the OntoQL associated exploitation language in order to support operations creation and invocation. The capability to create an operation is shown in the following example:

```
CREATE OPERATION unionOf
INPUTS REF #OWLClass ARRAY
OUTPUT REF #OWLClass
IMPLEMENTED WITH WEB_SERVICE unionOfImpl
namespace = "http://webservices.lias-lab.fr"
url = "http://localhost:8787/Services"
operation = "OWLUnionOfClasses"
```

This statement creates an operation `unionOf` that has an `OWLClass` array as input and returns an `OWLClass`. This operation is associated to an implementation `unionOfImpl` which is a web service that is described by some properties that permit its execution. Moreover, we have extended the query part of the OntoQL language with the capability to invoke operations.

```
CREATE #OWLClass SchoolMember
AS unionOf (Professor, Student)
```

This statement invokes the operation previously created to compute the creation of the new class `SchoolMember` from `Professor` and `Student` classes. Notice that the way the operation is implemented is completely hidden to the OntoQL user.

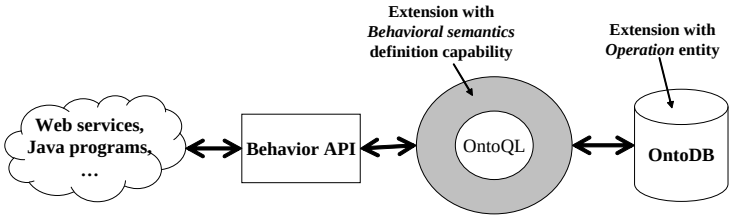

Fig. 8. The OntoDB/OntoQL behavior API

From a prototyping point of view, we have set up an application programming interface (Figure 8) that links the OntoDB/OntoQL world to the external world. Thus, this API makes data correspondences between these two worlds and invoke services and programs and return serialized (well formated) data.

We have defined our approach to extend the meta-schema part in order to handle operators within the OntoDB OBDB. This novel approach supports the definition of operations at runtime. In the remainder of this paper, we show how this extension can be exploited in order to define a methodology of OBDB design, more precisely for encoding non-canonical concepts.

5 Application: an ontology-based database design methodology

Since an OBDB is a database, it should be designed according to the classical design process dedicated to the development of databases, identified in the ANSI/X3/SPARC architecture. However, when exploring the database literature, we figure out that most of the research efforts were concentrated on the physical design phase, where various storage models for ontological data were given. Rdfsuite [1], Jena [9], OntoDB [14], Sesame [7], Owlgres [27], SOR [18], Oracle [12], etc. are examples for these systems. The fact that OBDB become mature, the proposition of a concrete design methodology as in traditional databases becomes a crucial issue. This development needs to follow the main steps of traditional database design approaches: *conceptual*, *logic* and *physical* designs. In [10], we proposed a five steps methodology for designing OBDB. It starts from a conceptual model to provide logic and physical models following five steps method [10]. In this work, we identify two types of ontological classes: (1) canonical (primitive) and (2) non-canonical (derived) classes. For the first type, we proposed a complete mechanism to manage canonical concepts structure and instances. On the contrary, for non-canonical concepts, we only proposed to place classes in a subsumption hierarchy and represent non-canonical instances using views after a step of inferring on the ontology achieved by a semantic reasoner. This proposition lacks an OBDB-integrated mechanism for representing non-canonical classes structure and instances views. Thus, we propose in this section, to extend OBDBs with a generic support of non-canonical concepts that could be used to complete this approach.

5.1 A design methodology

In this subsection, we introduce essential concepts to facilitate the understanding of our OBDB design methodology. Then, we describe the proposed methodology with its limitations. Finally, we show the interest of the OBDB extension we have achieved through a use case.

Ontological dependencies. The distinction between canonical and non-canonical concepts leads to define dependency relationships. As concepts include properties and classes, two categories of dependencies are identified: (1) Classes dependencies and (2) Properties dependencies.

1. **Classes dependencies**. Two types of classes dependencies may be distinguished: (1) Instance Driven Classes Dependencies (IDD^C) and (2) Static Dependencies (SD). In the first type, a functional dependency among two concepts C_1 and C_2 ($C_1 \rightarrow C_2$) exists if each instance of C_1 determines one and only one instance of C_2. [26] proposed an algorithm to discover functional dependencies (FD) among concepts of an ontology that exploits the inference capabilities of DL-Lite. For instance, if we consider a role

mastersDegreeFrom with a domain and a range respectively the *Person* and *University* classes, the FD $Person \rightarrow University$ is defined. It means that the knowledge of a person with a valued property *masterDegreeFrom* determines a knowledge of one instance of *University* class. In the second type, SD are defined between classes based on their definitions. A *SD* between two concepts C_i and C_j ($C_i \longmapsto C_j$) exists if the definition of C_i is available then C_j can be derived. Note that this definition is supported by a set of OWL[5] constructors (`owl:unionOf`, `owl:intersectionOf`, `owl:hasValue`, etc). For example, if we consider a property *level* having as domain the *Student* class, a class *MasterStudent* may be defined as a *Student* enrolled in the master level ($MasterStudent = \exists\ level\{master\}$; Domain $(master) = Student$). Therefore, the dependency $Student \longmapsto MasterStudent$ is obtained. It means that the knowledge of the whole instances of the *Student* class determines the knowledge of the whole instances of the *MasterStudent* class.

2. **Properties dependencies**. As in traditional databases, functional dependencies between properties have been identified in the ontology context [3, 8]. In [8], authors proposed a formal framework for handling FD constructors for any type of OWL ontology. Three categories of FDs are identified. In [3], we supposed the existence of FD involving simple properties of each ontology class. For instance, if we consider the properties $idProf$ and *name* having as domain the $Professor$ class and describing respectively the professor identifier and name, the FD $idProf \rightarrow name$ may be defined. It means that the knowledge of the value of the professor identifier $idProf$ determines the knowledge of a single value of *name*.

Methodology description In [10], we proposed a methodology which evolves from, and extends, the traditional database design process. It starts from a conceptual model to provide logic and physical models following a five step method as described in figure 9. In the first step, designer extracts a fragment of an, assumed available, domain ontology $\mathcal{O}$ (called local ontology(LO)) according to his/her requirements. The LO plays the role of the *conceptual model* (CM). This resulting local ontology is then analyzed automatically to identify canonical (CC^{LO}) and non-canonical classes (NCC^{LO}) by exploiting classes dependencies (Step 2). Based on the obtained CC^{LO} and NCC^{LO}, two further steps are defined in parallel. The first one concerns the placement of the NCC^{LO} in the OBDB taking into account the subsumption hierarchy of classes. The complete subsumption relationship for the ontology classes is produced by a reasoner such as Racer[6], Pellet [4], etc. (Step3). The second one describes the generation of the normalized logical model for each ontological class (Step 4) where:(i) a set of normalized tables (3NF) are generated for each CC_i (ii) a relational view is associated to each $CC_i \in CC^{LO}$ and (ii) a class view (a DL expression) on the canonical class(es) is associated to each $NCC_i \in NCC^{LO}$. Once the nor-

[5] http://www.w3.org/TR/owl-guide/
[6] http://www.racer-systems.com/

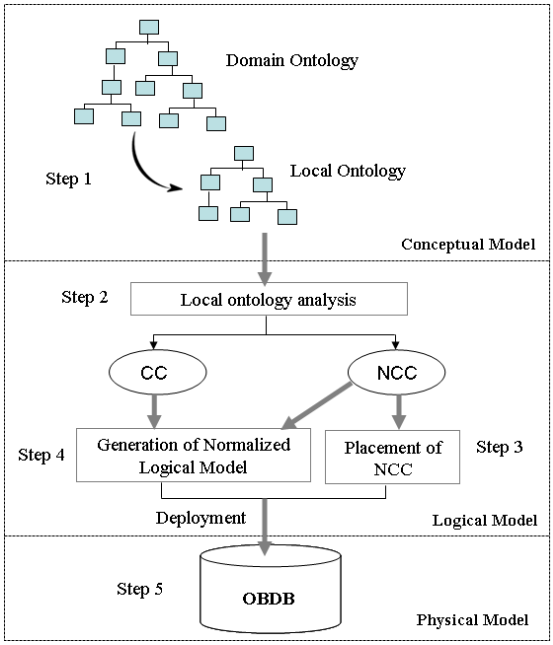

Fig. 9. OBDB design approach

input : $\mathcal{O}$: a domain ontology ;
output: *normalized OBDB*

Extract the local ontology LO;
Analyse the LO and identify CC^{LO} and NCC^{LO};
Place the NCC^{LO} in the appropriate subsomption hierarchy using an external reasoner;
foreach $CC_i^{LO} \in CC^{LO}$ **do**
  Generate the normalized tables (3NF);
  Generate a relational view defined on these tables;
end
foreach $NCC_i^{LO} \in NCC^{LO}$ **do**
  Generate a class view (a DL expression) on its related canonical class(es) ;
end
Choose any existing database architecture ;
Deploy ontological data;

Algorithm 1: OBDB design algorithm

malized logical model is obtained and NCC^{LO} are placed in the subsumption relationship, the database administrator may choose any existing database architecture offering the storage of ontology and ontological data (Step5) for making

persistent data describing meta-model, model and instances. The algorithm 1 summarizes these steps.

In our approach:

- we ask designer to be familiar with the use of reasoners in order to establish the complete subsumption relationships between ontological classes;
- once the class hierarchy is persisted to the target database, no updates can be made;
- no detail has been given for class views computation.

The defined approach has been hard encoded in the OBDB database management system. In next subsection, we show how the defined algorithm of the methodology can be triggered from the OBDB in a dynamic way.

5.2 Exploitation of the OntoDB extension to support OBDB design methodology

In this subsection, we show how the proposed methodology to design ontology-based databases taking into account (1) the different phases of classical design approach and (2) offering the definition of the behavioral semantics of model elements can be handled by the approach defined in section 4. We use OntoDB as storage model architecture for our *physical design phase*.

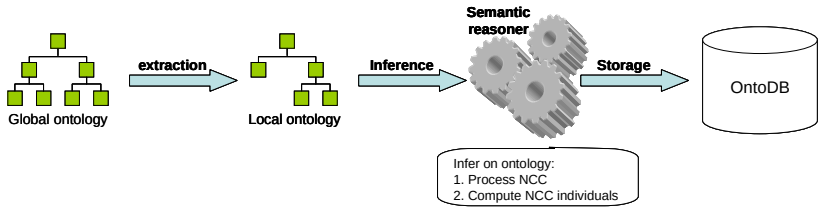

Fig. 10. Our initial OBDB design methodology

Basically, the OBDB design methodology use reasoners on ontologies to infer the non-canonical concepts instances before building the logical model (Figure 10).

By exploiting the support of non-canonical concepts in OBDB design methodology defined in section 4, we offer a permanent availability with operators to ontologies (Figure 11). Indeed, if the ontology is updated, we can use the defined operators to infer on the ontology and/or calculate the new eventual derived concepts. This approach avoids us to restart the OBDB design process in order to rebuilt the logical model.

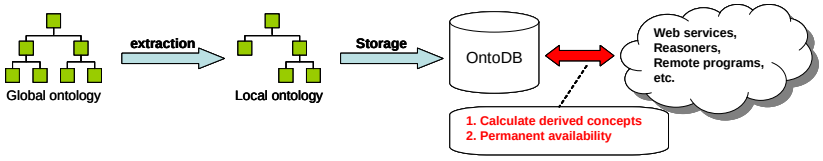

Fig. 11. OBDB design methodology supporting non-canonical concepts

Explicit classes dependencies with operations. To show a real use case of the proposed methodology, the extension of the initial ontology model stored in the meta-schema part of OntoDB is required. Thus, we first enrich the meta-schema to handle (a) class dependencies, and (b) operations expressing behavioral semantics of non-canonical concepts.

In order to handle class dependencies in OntoDB, we extend the meta-schema part:

```
CREATE ENTITY #CLeftPart (#itsClasses REF(#OWLClass) ARRAY);
CREATE ENTITY #CRightPart (#itsClass REF(#OWLClass));
CREATE ENTITY #CDependency (#rightPart REF(#CRightPart), #leftPart REF(#CLeftPart),
#operator REF(#Operation));
```

These statements extend the meta-schema part of OntoDB with three entities in order to handle class dependencies including the left parts, the right part and the operation used to compute the right part class. Indeed, operations help us to define precisely the nature of the defined classes dependencies and indicate the operators used to compute the resulting class of the dependency. As example, the next statement expresses and stores the classes dependency $(Professor, Student \rightarrow SchoolMember)$. It states that `Professor` and `Student` classes determine together the `SchoolMember` class using the `unionOf` operator.

```
CREATE #CDependency (#leftPart (Professor, Student), #rightPart (SchoolMember),
#operator (unionOf));
```

This dependency will be used to derive the non-canonical `SchoolMember` class the union of `Professor` and `Student` classes.

Explicit properties dependencies with operations. To support functional dependencies, the meta-schema part has to be extended with entities storing functional dependencies.

```
CREATE ENTITY #FLeftPart (#itsProps REF(#OWLProperty) ARRAY);
CREATE ENTITY #FRightPart (#itsProp REF(#OWLProperty));
CREATE ENTITY #FDependency (#rightPart REF(#FRightPart), #leftPart REF(#FLeftPart),
#operator REF (#Operation), #itsClass REF(#OWLClass));
```

Similarly to classes dependencies, the statements above extend the meta-schema with three entities to handle functional dependencies between properties of a given class. A functional dependency is characterized by a left part which is defined by one or many properties, a right part property, an eventual operator to calculate the right part property if it is a derived property and the class on which the dependency is defined.

The statements below create two functional dependencies. The first one expresses the dependency between `idProf` and `name` properties of the `Professor` class, and the second one expresses the dependency between `birthday` and `age` properties of `Professor`. In the second dependency, the `age` property is computed using the `calculateAge` operator. This dependency will be used to derive a 3NF schema of tables associated to the `Professor` class.

```
CREATE #FDependency (#leftPart (idProf), #rightPart (name), #itsClass (Professor));
CREATE #FDependency (#leftPart (birthday), #rightPart (age), #operator (calculateAge),
#itsClass (Professor));
```

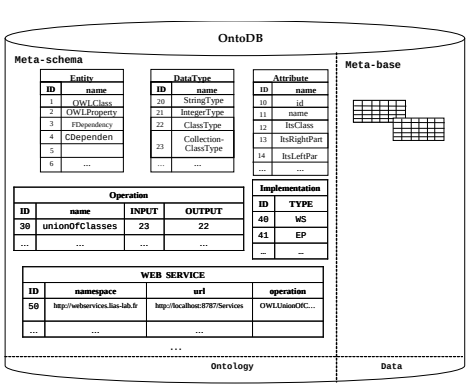

Fig. 12. Meta-schema depoyment

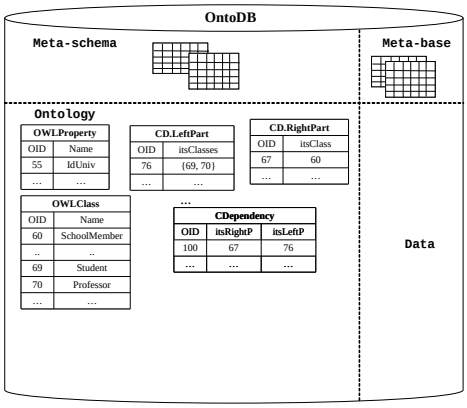

Fig. 13. Ontology deployment

Figure 12 shows the meta-schema deployment in which the ontology model and operations are stores, and Figure 13 shows the deployment of the ontology in which ontologies together with classes and properties dependencies are stored.

Generating the logical model of an OBDB. Once operations are created and stored in the OBDB, and classes and properties dependencies are expressed, we become able to generate a complete logical model of the OBDB to design integrating both a 3NF schema for each canonical class and a class view for each non-canonical class.

To compute non-canonical concepts, we invoke an existing operation. For example, the `SchoolMember` non-canonical class is computed using the following

statement:

```
CREATE #OWLClass SchoolMember
AS unionOf (Professor, Student)
```

To generate the logical model of the OBDB we invoke an operation that execute a program implementing the algorithm 1. The logical model of the OBDB to design could be obtained with the statement below:

```
CREATE #LogicalSchema
AS algorithm();
```

With regard to the `SchoolMember` class, a class view is computed based on its definition ($SchoolMember \equiv Professor \cup Student$). This view is associated to the non-canonical concept:

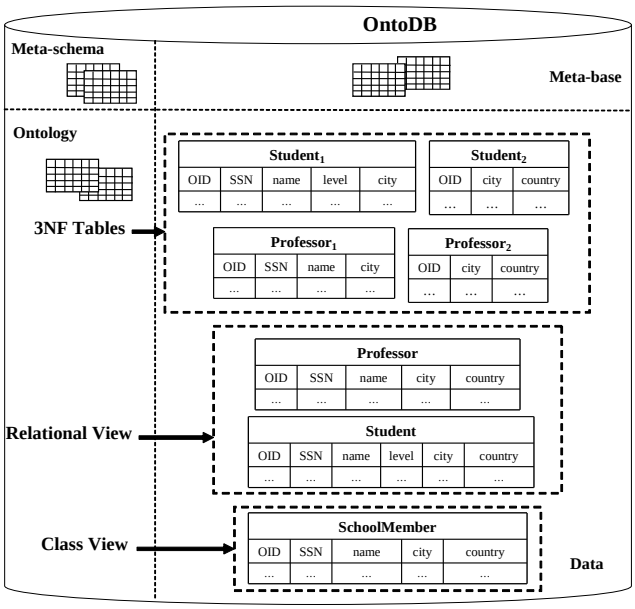

Fig. 14. Data deployment

Figure 14 shows an example of the deployment of the generated normalized logical model of the `Professor`, `Student` and `SchoolMember` classes in OntoDB.

6 Conclusion

In this paper, we have shown that existing OBDB did not address non-canonical concepts (NCC) as well as it is done with ontology editors evolving in main

memory. Indeed, OBDB use some mechanisms that are not sufficient, adapted or suitable all the time. Thus, we have proposed a generic solution to handle NCC by introducing dynamically new operators that can be implemented in different manners (e.g. web services or external programs). This extension will permit to OBDB to use internal mechanisms as well as external ones to handle NCC. Besides, this extension gives to OBDB a wide coverage of ontologies. In order to validate our approach, we have improved a methodology to design OBDB where ontologies were basically inferred by reasoners before building the logical model. As perspectives, we expect to use this approach to perform model transformations and derivations in a persistent environment and to address scalability issues. We also expect to complement OBDB architectures with constraints definition and checking and associate automatic reasoning to OBDB since they may be encoded by callable operations or constraints solvers.

References

1. S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing voluminous RDF description bases. In *Proceedings of the Second International Workshop on the Semantic Web (SemWeb)*, 2001.
2. Youness Bazhar. Handling behavioral semantics in persistent meta-modeling systems. In *Proceedings of the Sixth IEEE International Conference on Research Challenges in Information Science, RCIS*, Valencia, Spain, May 2012.
3. L. Bellatreche, Y. Aït Ameur, and C. Chakroun. A design methodology of ontology based database applications. In *Logic Journal of the IGPL*, 2010.
4. S. Evren P. Bijan. Pellet: An owl dl reasoner. In *International Workshop on Description Logics (DL2004)*, pages 6–8, 2004.
5. Alex Borgida and Ronald J. Brachman. Loading data into description reasoners. *SIGMOD Record*, 22(2):217–226, 1993.
6. Dan Brickley and R V. Guha. *RDF Vocabulary Description Language 1.0: RDF Schema*. World Wide Web Consortium, 2004. `http://www.w3.org/TR/rdf-schema`.
7. J. Broekstra, A. Kampman, and F. V. Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International Semantic Web Conference*, pages 54–68, 2002.
8. J. P. Calbimonte, F. Porto, and C. Maria Keet. Functional dependencies in owl abox. In *Brazilian Symposium on Databases (SBBD)*, pages 16–30, 2009.
9. Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (WWW'04)*, pages 74–83, New York, NY, USA, 2004. ACM Press.
10. C. Chakroun, L. Bellatreche, and Y. Aït Ameur. The role of class dependencies in designing ontology- based databases. In *The 7th International IFIP Workshop on Semantic Web and Web Semantics*, pages 444–453, 2011.
11. Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *Proceedings of the 31st international conference on Very Large Data Bases (VLDB'05)*, pages 1216–1227, 2005.

12. S. Das, E. I. Chong, G. Eadon, and J. Srinivasan. Supporting ontology-based semantic matching in rdbms. In *VLDB*, pages 1054–1065, 2004.
13. M. Dean and Schreiber. Web Ontology Language Reference. *W3C Recommendation (2004)*, 2004.
14. Hondjack Dehainsala, Guy Pierra, and Ladjel Bellatreche. Ontodb: An ontology-based database for data intensive applications. In *Proc. of the 12th Int. Conf. on Database Systems for Advanced Applications (DASFAA'07). LNCS*. Springer, 2007.
15. Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk RDF Storage. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PPP'03)*, pages 1–15, 2003.
16. S. Jean, Y. Aït Ameur, and G. Pierra. Querying ontology based databases - the ontoql proposal. In *Proceedings of the 18th International Conference on Software Engineering & Knowledge Engineering (SEKE'2006)*, pages 166–171, 2006.
17. Stéphane Jean, Yamine Aït Ameur, and Guy Pierra. Querying ontology based database using ontoql (an ontology query language). In *OTM Conferences (1)*, pages 704–721, 2006.
18. J. Lu, L. Ma, L. Zhang, J. S. Brunner, C. Wang, Y. Pan, and Y. Yu. Sor: A practical system for ontology storage, reasoning and search. In *VLDB*, pages 1402–1405, 2007.
19. Frank Manola and Eric Miller. *RDF Primer*. World Wide Web Consortium, 2004. `http://www.w3.org/TR/rdf-primer`.
20. Jing Mei, Li Ma, and Yue Pan. Ontology query answering on databases. In *Proceedings of the 5th International Semantic Web Conference (ISWC'06)*, pages 445–458, 2006.
21. Z. Pan and J. Heflin. Dldb: Extending relational databases to support semantic web queries. In *The First International Workshop on Practical and Scalable Semantic Systems*, 2003.
22. M Jae Park, Ji Hyun Lee, Chun Hee Lee, Jiexi Lin, Olivier Serres, and Chin Wan Chung. An efficient and scalable management of ontology. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07)*, volume 4443 of *Lecture Notes in Computer Science*. Springer, 2007.
23. Johan Petrini and Tore Risch. SWARD: Semantic Web Abridged Relational Databases. In *Proceedings of the 18th International Conference on Database and Expert Systems Applications (DEXA'07)*, pages 455–459, 2007.
24. Guy Pierra. Context Representation in Domain Ontologies and its Use for Semantic Integration of Data. *Journal Of Data Semantics (JODS)*, X:34–43, 2007.
25. Guy Pierra and Eric Sardet. *ISO 13584-32 Industrial automation systems and integration Parts library Part 32: Implementation resources: OntoML: Product ontology markup language*. ISO, 2010.
26. O. Romero, D. Calvanese, A. Abelló, and M. Rodriguez-Muro. Discovering functional dependencies for multidimensional design. In *DOLAP*, pages 1–8, 2009.
27. M. Stocker and M. Smith. Owlgres: A scalable owl reasoner. In *The Sixth International Workshop on OWL: Experiences and Directions*, 2008.
28. Raphael Volz, Steffen Staab, and Boris Motik. Incrementally Maintaining Materializations of Ontologies Stored in Logic Databases. *Journal of Data Semantics II*, 3360:1–34, 2005.