Ontological Concepts Dependencies Driven Methodology to Design Ontology-Based Databases

Chedlia Chakroun¹, Ladjel Bellatreche¹, and Yamine Ait-Ameur¹

LISI/ENSMA - Poitiers University
Futuroscope, France
(chakrouc, bellatreche, yamine)@ensma.fr

Abstract. Nowadays, a large number of applications are producing and manipulating ontological data. Managing a huge amount of these data needs the development of scalable solutions. Ontology-based database (OBDB) is one of these solutions. An OBDB stores in the same repository ontological data and the ontology describing their meanings. Several architectures supporting these OBDB were proposed by academicians and industrial editors of DBMS. Unfortunately, there is no available methodology for designing such OBDB. To overcome this limitation, this paper proposes to scale up the traditional database design approaches to OBDB. Our approach covers both conceptual and logical modeling phases. It assumes the availability of a domain ontology composed by canonical and non canonical concepts. Dependencies between properties and classes are captured and exploited to define a two levels normalized logical model without redundancy (canonical): at class level and at class properties level. A prototype implementing our methodology on the OBDB OntoDB is outlined.

1 Introduction

Roughly speaking, ontologies [16] have been introduced in information systems as knowledge models that provide with definitions and descriptions of the concepts of a target application domain [21, 24, 28, 26]. Nowadays, we assist to a spectacular explosion of ontological data manipulated and produced by a large number of applications in various domains such as scientific computation, Web service, engineering, social networks, etc. This is mainly due to three factors: (i) the development of domain ontologies, (ii) the availability of commercial and open source software tools for building, editing and deploying ontologies and (iii) the existence of ontology model formalisms (OWL, PLIB, etc.) which have considerably contributed to the emergence of ontology based applications exploiting ontological data. Faced to this situation, managing, storing, querying these data and reasoning on them require the development of software tools capable of handling these activities. Moreover, due to the wide usage of such data, the need of a systematic and consistent development process appeared, taking into account the central objectives of persistence, scalability and high performance of
the applications. Persistence of ontological data (also called *ontology store* [11]) was advocated by academic and industrial researchers. As a consequence, a new type of databases, named *ontology-based databases* (OBDB) dedicated to store, manage and exploit ontological data emerged. An OBDB is a database model that allows both ontologies and their instances to be stored (and queried) in a single and homogeneous repository [13, 20]. Since an OBDB is a database, it should be designed according to the classical design process dedicated to the database development, identified in the ANSI/X3/SPARC architecture. At the origin, this architecture does not recommend the use of ontologies as a domain knowledge model, but it refers to the notion of dictionary or catalog. In [14], an extension of this architecture giving ontologies their right place during the life cycle of database scheme process design has been proposed. But, the reality is different. Indeed, 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 [3], Jena [9], Ontodb [13], Sesame [7], Owlgres [27], SOR [20], Oracle [12], etc. are examples of such systems. Note that each system has its own storage model. They are built according to three main architectures. In the first one, the ontology and its associated data are stored in a RDF triples structure: *(subject, predicate, object)*. In this architecture, there is no identified separation between ontology and data [9]. In the second architecture, ontology and its ontological data are stored independently into two different schemes. Sesame is an example of such architecture. The storage model used for the ontology is based on RDFS, whereas data may be represented using different storage models: (1) a unique table of triples containing extensions of all concepts of the ontology. (2) a unary distinct table

![Fig. 1. Big picture on designing OBDBs](image)
for each class of the ontology and a binary table for each property of the ontology. Observe that the management of ontology and data parts is different. This storage model scales quite well, especially, when queries refer to a small number of properties [1]. These two architectures share a common property: they both hardly encode the ontology model (RDF or RDFS). Unlike logical models in the ANSI/X3/SPARC architecture, their physical structure is static and does not evolve according to the stored ontological data model. The third architecture extends the second one, by adding a new part, called, the meta schema part. As a result, it contains three parts: meta schema, ontology and data. The presence of the meta schema offers flexibility of the ontology part modeling, since it is represented as an instance of the meta schema. Moreover, flexibility is also offered in the design of the ontology logical model. This approach offers a generic access to both ontology and data. Note that different storage models may be used for the data part. For instance, OntoDB [13] uses a horizontal storage model, where a single table is associated to each class of the ontology with one column per each used property. This architecture is more efficient than the previous ones for certain types of queries [13]. To facilitate the exploitation of these OBDB systems, different query languages were proposed (OntoQL [17], RQL [19], SPARQL [23], etc.). Figure 1 depicts the three addressed OBDB architectures.

| TABLES |
|--------|--------|---------|
| SUBJECT | PREDICATE | OBJECT |
| ULCity   | aCity   | ULCity  |
| PULCity  | aCity   | PULCity |
| PublicUL | aCity   | PublicUL|
| ...      | ...     | ...     |
| ULCity   | aCity   | category |
| ULCity   | aULmage | ULmage   |
| ULCity   | aString | ULString |
| ULCity   | aName   | ULCity   |
| PublicUL | aName   | PublicUL|
| PublicUL | aType   | PublicUL|
| PublicUL | aName   | PublicUL|
| PublicUL | aName   | PublicUL|
| ...      | ...     | ...     |

Fig. 2. Intra table redundancy

The fact that OBDB become mature and since they have been used in several projects, the proposition of a concrete design methodology, like in traditional databases, becomes a crucial issue for companies. The development of such a methodology needs to follow the main steps of traditional database design approaches: conceptual, logical and physical. Actually, if one wants to design an OBDB from a given domain ontology, she/he should perform the following tasks.
(i) choose her/his favorite architecture, (ii) identify the relevant storage models, (iii) establish mapping between ontology concepts and the target entities of the chosen storage model. This design procedure has several drawbacks: (a) redundancy intra and inter relations (Figure 2), (b) designers need to deeply understand the physical storage models and architectures to develop their own applications, (c) lack for data access transparency, since users have to know the physical storage model to query the final OBDB.

Recently, some research studies recommending the use of ontologies to design traditional databases and data warehouses raised up in the literature. The similarities between ontologies and conceptual models [26] and the reasoning capabilities offered by ontologies have motivated such studies. Indeed, [28] used linguistic ontologies to conceptually design traditional databases while [21, 24] use ontology to design multidimensional data warehouses. The reader may observe that in the last decade, data warehouses faced the same phenomena as for OBDB, where most of the studies focused on the physical design [10]. So, although conceptual design and requirement analysis are two of the key steps within the database and data warehouse design processes [15], they were partially neglected in the first era of OBDB.

In the classical databases, the redundancy is usually resolved by normalizing the logical model obtained from conceptual model. This normalization is performed thanks to the exploitation of the available functional dependencies (FD) between properties. Recently, a couple of studies enriched ontology models by FD defined on properties and classes [5, 8, 25]. The similarities between conceptual models and ontologies and their support of functional dependencies motivate us to develop a complete methodology for designing OBDB. Keeping in mind the ANSI/X3/SPARC model, this paper gives answers to the lack of design methodology of OBDB, including conceptual, logical, physical phases and offering a transparent access to data via the classes of the domain ontology.

This paper is divided into five sections: section 2 describes the basic concepts related to dependencies between ontological concepts and a formal model of ontology. The main steps of our methodology are presented in section 3. Section 4 validates our methodology using the Lehigh University Benchmark ontology. Finally, section 5 concludes the paper by summarizing the main results and suggesting future work.

2 Basic Concepts and Formalization

In this section, we present a taxonomy of ontologies, dependencies between the ontological concepts and a formal model of ontologies. Our classification is driven by information systems modeling.

2.1 Taxonomy of ontologies

In [18], a taxonomy of ontologies, namely the onion model, has been proposed. It considers the concept with a set of properties as the basic notion for ontology
design rather than the term as in semantic web based approaches. The three layers of the onion model are: (1) Conceptual Canonical Ontologies (CCO) can be considered as shared conceptual models. They contain the core classes concepts. (2) Non Conceptual Canonical Ontologies (NCCO) extend CCO by allowing the designers to introduce derived classes and concepts. NCCO give the same functionalities offered by views in databases. Therefore, a non canonical concept can be seen as a virtual concept defined from canonical concepts. (3) Linguistic Ontologies (LOs) are the upper extension. They may be used to document existing databases or to improve the database-user dialog or to support multi-lingual applications. This taxonomy helps designers to identify canonical and no canonical classes of a given domain ontology. The following example shows how this taxonomy is set up for this purpose.

Example 1. Figure 3 represents an extended fragment of the Lehigh University Benchmark (LUB)\(^1\), where the concept University defined as the union of two canonical concepts: PublicUniversity (PU) and PrivateUniversity (PRU) \((\text{University } \equiv \text{PU } \cup \text{PRU})\) is added. Based on the definition of the ontology concepts, canonical (CC) and non canonical classes (NCC) are identified as follows:

\[
NCC_{\text{LUB}} = \{\text{University, Student, Employee, MasterCourse}\}, \quad CC_{\text{LUB}} = C_{\text{LUB}} - NCC_{\text{LUB}},
\]

where \(C_{\text{LUB}}\) represents all classes of LUB. Starting from the fact that non canonical concepts are directly or indirectly defined from canonical ones, a dependency relation between canonical and non canonical concepts may exist. For instance, if we consider the previous definition of the University concept, a dependency relation is defined as: \((PU, PRU) \rightarrow University\). Note that other types of dependencies between ontological concepts detailed in the next section have been studied in the literature.

Fig. 3. Extended LUB Ontology

\(^1\) the ontology and its class definitions described in OWL are available: http://www.lehigh.edu/zhp2/2004/0401/univ-bench.owl
2.2 Dependencies between Ontological Concepts

In this section, we show the interest of capturing dependencies between ontological concepts on designing OBDB. Our analysis on dependencies identified in the ontology context [5, 8, 25] gives rise to the following classification: instance driven dependencies (IDD) and static dependencies (SD). IDD is quite similar to functional dependencies (FD) in traditional databases [2]. IDD may concern either properties [8] or classes [25] of a given ontology. Calbimonte et al. [8] proposed a formal framework for handling and supporting FD constructors for any type of OWL ontology. They distinguished three categories of FDs: classical, keys and explicit dependencies. Classical and keys correspond to the traditional FD, whereas the last one is a particular case of tuple generating dependency. In [5], we supposed the existence of FD involving simple properties of each ontology class. For instance, if we consider a class Student (Figure 3) with a set of properties id, age and name, a FD may be defined as: (id → name, age).

In [25], the authors proposed an algorithm to discover FD among concepts of a domain ontology that exploits the inference capabilities of DL-Lite. A FD among two concepts $C_1$ and $C_2$ ($C_1 → C_2$) exists if each instance of $C_1$ determines one and only one instance of $C_2$. The FD between concepts is an extension of classical FD between properties defined in databases. The identification of FD is performed by the means of functional roles between the concerned concepts. For instance, if we consider a role MasterDegreeFrom with a domain and range Person and University, the following FD is defined: Person → University.

SD are defined between classes based on their definitions (see Example 1). A SD between two concepts $C_i$ and $C_j$ (denoted by $C_i → C_j$) exists if the definition of $C_i$ is available then $C_j$ can be derived. The OWL2 constructors that we consider in this paper are: hasValue, unionOf, intersectionOf, allValuesfrom. For example, hasValue allows specifying classes based on the existence of particular property values. If we consider a class University having status as one of its properties, a class PublicUniversity may be defined as an University with a public status ($\text{PublicUniversity} ≡ \exists \text{status}. \{\text{Public}; \text{Domain}($\text{status}$) = \text{University})$. Based on this definition, the following dependency $\text{University} → \text{PublicUniversity}$ is obtained. Figure 4 summarizes the different dependencies identified in the ontology context.

2.3 A Formal Model for Ontologies

Formally, an ontology $\mathcal{O}$ may be defined as follows: $< \mathcal{C}, \mathcal{P}, \text{Applic}, \text{Sub}, \text{FDP}, \text{CD} >$, where:

- $\mathcal{C}$ is the set of the classes used to describe the concepts of a given domain.
- $\mathcal{P}$ is the set of all properties used to describe the instances of $\mathcal{C}$.
- $\text{Applic}$ is a function defined as $\text{Applic} : \mathcal{C} → 2^\mathcal{P}$. It associates to each class of $\mathcal{O}$, the properties that are applicable for each instance of this class. Note that for each $c_i ∈ \mathcal{C}$, only a subset of $\text{Applic}(c_i)$ may be valued in a particular database, for describing $c_i$ instances.

Footnote: http://www.w3.org/TR/owl-guide/
– *Sub* is the subsumption relationship defined as \( Sub : C \rightarrow 2^C \), where for a class \( c_i \in O \), it associates its direct subsumed classes\(^3\). \( Sub \) defines a partial order over \( C \). In our model, two kinds of subsumption relationships are introduced: \( Sub = OOSub \cup OntoSub \), where:

* \( OOSub \) is the usual object oriented subsumption of single inheritance relationship. Through \( OOSub \), the whole set of applicable properties are inherited.

* \( OntoSub \) is a subsumption relationship without inheritance. Through \( OntoSub \) (also called *case-of* in the PLIB ontology model \([22]\), or defined as partial inheritance), whole or part of the whole applicable properties of a subsuming class may be explicitly imported by a subsumed class.

– *FDP* : a mapping from the powerset of \( P \) onto \( P \) \((2^P \rightarrow P)\) representing IDD defined on the applicable properties of \( P \) of a class in \( O \).

– *CD* : a mapping from the powerset of \( C \) onto \( C \) \((2^C \rightarrow C)\) representing either static or instance-driven dependencies

\[ c_1 \text{ subsumes } c_2 \iff \forall x \text{ instance of } c_2, x \text{ is instance of } c_1. \]

\(^3\) \( c_1 \) subsumes \( c_2 \) iff \( \forall x \) instance of \( c_2 \), \( x \) is instance of \( c_1 \).

3 Our proposal

Now, all ingredients to propose a complete methodology to design OBDB from an OWL domain ontology \( O \) respecting the above formal model are available.

3.1 Different Phases of our Methodology

The proposed methodology is inspired from the database design process. It starts from a conceptual model to provide logical and physical models. According to figure 5, our approach is a five steps method. It starts from the extraction of a local ontology (step 1) and then it identifies canonical and non canonical classes

Fig. 4. Dependencies Classification
by exploiting class dependencies (step 2). As a further step, it defines in parallel a placement of the NCC (step 4) and a logical model for each CC (step 3). Once these steps are performed, it becomes possible to generate a logical model for the OBDB as shown in figure 7 where: (i) a relational view is associated to each CC, (ii) a class view (a DL expression) on the CC is associated to each non canonical class.

Step 1. During this phase, the designer extracts a fragment of the domain ontology $O$ (that we call local ontology $LO$) according to her/his requirements. The $LO$ plays the role of conceptual model (CM). Three extraction scenarios may occur:

1. $LO = O$ means that $O$ covers all the designer requirements.
2. $LO \subset O$ means that $O$ is rich enough to cover the user requirements. The construction of $LO$ is performed using the operator $OntoSub$ (see section 2.3). This relationship is an articulation operator allowing to connect $LO$ to a $O$ while offering a large independence of $LO$. Through this relationship, a local class may import or map all or part of the properties that are defined in the referenced class(es).
3. $LO \supset O$ means that the ontology $O$ does not fulfill the whole designer requirements. In this case, designer extracts from the $O$ a fragment corresponding to her/his requirements and locally enriches it by adding new concepts/properties. Note that dependencies may exist between the new added and imported properties and concepts.
Step 2. Once LO is extracted, an analysis needs to identify canonical and non canonical classes is performed. Let $C^{LO}$, $CC^{LO}$ and $NCC^{LO}$ be, respectively, the set of all classes, canonical classes and non canonical classes of LO. Dependencies between different ontology classes may be represented by a directed graph $G : (C^{LO}, A)$, called dependency graph, where the nodes are $C^{LO}$, and an edge $a_k \in A$ between a pair of classes $c_i$ and $c_j$ ($\in C^{LO}$) exists, if a dependency between $c_i$ and $c_j$ has been established (Figure 6). This graph is used to determine the minimum closure-like classes that will represent the canonical classes. Our graph dependency is quite similar to functional dependency graph in the classical databases to generate minimum closure and normalize tables [4]. The sole difference is that we have classes as nodes whereas; in functional dependency graph we have attributes (properties) as nodes. Based on this similarity, we adapt [4]’s algorithm to generate our $CC^{LO}$ (see Algorithm 1). Note that this algorithm may generate different sets of canonical classes (when several candidate classes may occur) as shown in Figure 6. In this case, the designer may choose her/his relevant set.

![Class Dependencies](image)

Class Dependencies:
$C_2, C_3 \rightarrow C_1$ ; $C_1 \rightarrow C_3$ ; $C_3, C_4 \rightarrow C_4$ ; $C_9, C_8 \rightarrow C_7$

Class Identification:
Solution 1: $CC = (C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9)$ ; $NCC = (C_1, C_2, C_3)$
Solution 2: $CC = (C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9)$ ; $NCC = (C_1, C_2, C_3)$

![Fig. 6. Example of Class identification](image)

Step 3. Based on the obtained $CC^{LO}$ and $NCC^{LO}$, two scenarios may be distinguished:

1- $NCC^{LO} = \emptyset$: Only $CC^{LO}$ exists. Then, the FD defined on their properties are used for normalization and for the definition of their primary keys (for more details see [5]).

2- $CC^{LO} \neq \emptyset$ and $NCC^{LO} \neq \emptyset$. For each class in $CC^{LO}$, the same mechanism described in Step3.1 above is applied. Then, for each class in $NCC^{LO}$, a relational view is computed. For example, let $ncc_j \in NCC^{LO}$ be a non canonical class defined as the union of two canonical classes $cc_1(p_1, p_2, p_3)$ and $cc_2(p_1, p_2, p_4)$, a view corresponding to $ncc_j$ is defined as follows:
input: $G : (C^{LO}, A)$: a dependency graph $G$ with a set of classes $C^{LO}$ of $LO$ and a set of edges $A$; $Domain(G)$: set of nodes (classes) not having a predecessor node (class); $Range(G)$: set of nodes (classes) having at least a predecessor node (class); $S_{Min}$: set of classes representing one of the solutions of $CC^{LO}$; $SS_{Min}$: set of $S_{Min}$ representing all solutions of $CC^{LO}$; $A_i : (LC, RC)$: an edge $A_i \in A$ has a source $LC$ (set of classes) and a destination $RC$ (a class); $N_{Known}$: set of classes deduced from the knowledge of $S_{Min}$; $N_{Unknown}$: set of not deductible classes from the knowledge of $S_{Min}$; 

output: $SS_{Min}$

Init:
$Domain(G) \leftarrow LC_A \cup \{C^{LO} | C^{LO} \notin \{RC_A \cup LC_A\}\}$;
$Range(G) \leftarrow RC_A$;
$S_{Min} \leftarrow \{C^{LO} | C_i^{LO} \in Domain(G) \& C_i^{LO} \notin Range(G)\}$;
$N_{Known} \leftarrow S_{Min}$;
$SS_{Min} \leftarrow \emptyset$;

Canonical class generation;
if $S_{Min} = Domain(G)$ then
    $SS_{Min} \leftarrow SS_{Min} \cup S_{Min}$
    $G \leftarrow \emptyset$
else
    $SS_{Min} \leftarrow \text{Canonicity}(G, N_{Known}, S_{Min}, SS_{Min})$
end

Function Canonicity($G, N_{Known}, S_{Min}, SS_{Min}$);
while $LC_A \subset N_{Known}$ do
    if $RC_A \notin N_{Known}$ then
        $N_{Known} \leftarrow N_{Known} \cup \{RC_A\}$;
    end
    $G \leftarrow G - A_i$;
end
if $G = \emptyset$ then
    $SS_{Min} \leftarrow SS_{Min} \cup S_{Min}$;
else
    $N_{Unknown} \leftarrow \{C_i^{LO} | ((C_i^{LO} \in Domain(G)) \& (C_i^{LO} \notin N_{Known}))\}$
    while $N_{Unknown} \neq \emptyset$ do
        foreach $C_i^{LO} \in N_{Known}$ do
            $N_{Known} \leftarrow N_{Known} \cup \{C_i^{LO}\}$;
            $S_{Min} \leftarrow S_{Min} \cup \{C_i^{LO}\}$;
            $N_{Unknown} \leftarrow N_{Unknown} - \{C_i^{LO}\}$;
            $SS_{Min} \leftarrow \text{Canonicity}(G, N_{Known}, S_{Min}, SS_{Min})$;
            $S_{Min} \leftarrow S_{Min} - \{C_i^{LO}\}$;
            $N_{Known} \leftarrow N_{Known} - \{C_i^{LO}\}$;
        end
    end
end

Algorithm 1: Canonicity Algorithm
((Select \(p_1, p_2\) From \(cc_1\)) Union (Select \(p_1, p_2\) From \(cc_2\))).

One of the advantages of using views to represent non canonical classes is to ensure the transparency to access data. A designer may query an OBDB via these classes without worrying about the physical implementation of those classes. Figure 7 summarizes these steps.

**Step 4.** The ontology classes may be defined without specifying the subsumption relationship. Thus, we propose to store all ontology classes regardless of their types (CC and NCC) 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, Pellet [6], etc. For example, the class *PublicUniversity* (*PublicUniversity* \(\equiv \exists\text{status.}\{\text{Public}\})*, will be a subclass of the *University* class.

**Step 5.** Once the normalized logical model obtained and NCC placed in the subsumption relationship, the database administrator may choose any existing database architecture offering the storage of ontology and ontological data.

![Fig. 7. Generation of normalized logical model.](image)

### 4 Validation of our Design Methodology.

In this section, we propose a validation of our design approach. We use OntoDB [13] as storage model architecture for our *physical design phase* (Step 5). OntoDB is chosen for three main reasons: (i) it belongs to the third architecture (Section 1) that allows us to enrich the meta schema by dependencies defined on classes and properties. (ii) OntoDB outperforms most of existing systems belonging to architecture I and II [13], (iii) A prototype is available, it has been used in several industrial projects (*Peugeot and Renault French Car companies, French Institute of Petroleum, etc.*) and it is associated with a query language called **OntoQL** [17] defined as an extension of SQL for exploiting both data and their semantics stored in an OBDB. OntoQL has the capability to express queries on data independently of their schemes and allows users to query ontologies,
instances and both ontologies and data. Our two algorithms for determinating canonical and non canonical classes and for normalizing each class using functional dependencies defined on its properties were encoded within Java. Note that these algorithms are based on graph dependencies [4].

4.1 Validation Process Main Steps

To validate our proposal, we follow the different steps of our proposal. Before we start the application of our approach, we show how dependencies can be made persistent in the OntoDB OBDB of type III architecture. The meta schema of OntoDB contains two main tables Entity and Attribute encoding the meta-meta model level of the Meta-Object Facility (MOF) architecture. To conduct our validation, the support of the ontological concept dependencies in the initial kernel of OntoDB is required. Faced to this requirement, we extended the meta schema of OntoDB by incorporating dependencies defined on properties and classes. First, we develop a meta model describing both dependencies. Figure 8 shows the UML model of this meta model called (MMPCD). Once this model is encoded, the obtained extended meta schema gives us the possibility to store (in a persistent form) ontologies with their dependencies (Figure 8 b). The following OntoQL statements instantiate the two main tables Entity and Attribute with dependency description.

```
CREATE ENTITY #C.Left.Part (#its.LeftPart.Classes REF(#Class) ARRAY)
CREATE ENTITY #C.Right.Part (#its.RightPart.Class REF(#Class))
CREATE ENTITY #CD ( #its.C.RightPart REF(#C.Right.Part),
  #its.C.LeftPart REF(#C.Left.Part))
CREATE ENTITY #P.Right.Part(#its.RightPart.Prop REF (#Property));
CREATE ENTITY #P.Left.Part(#its.LeftPart.Prop REF (#Property)ARRAY);
```

Fig. 8. (a) MMPCD of meta model, (b) Extended OntoDB by PFDs and CDs
CREATE ENTITY #PFD (#itsClass REF (#Class), #its.P.Right.Part REF (#P.Right.Part), #its.P.Left.Part REF(#P.Left.Part));

**Step 1.** Once the meta schema is extended, the local ontology defined from user’s requirements is created in the the ontology part of OntoDB architecture. The following OntoQL statements create `PublicUniversity` class (defined in Example 1) with the following properties: `IdUniv`, `Name`, `City`, `Region`, `Country` and `UniversityStatus`.

CREATE #Class PublicUniversity(DESCRIPTOR (#name[en] = 'PublicUniversity') PROPERTIES( IdUniv INT, Name STRING, City STRING, Region STRING, Country STRING,UniversityStatus STRING))

After creating the structure of a class, their dependencies defined on properties and classes should be attached. Let us assume that the following dependency between two classes `University` and `PublicUniversity` exists (`University` →→ `PublicUniversity`), `PublicUniversity` class defined as an `University` having `Public` as value of its property `UniversityStatus`). This dependency is defined by the following OntoQL statement:

Insert Into #CD (#its.C.RightPart, #its.C.LeftPart) Values((Select #oid from #Class c Where c.#name='PublicUniversity'), (Select #oid from #Class c Where c.#name='University'))

Instantiation of dependencies between properties is handled in a similar way. For instance, if a dependency between two properties `IdUniv` and `Name` of `PublicUniversity` exists, it is defined by the following statement:

Insert Into #PFD (#itsClass, #its.P.Right.Part, #its.P.Left.Part) Values ((Select #oid from #Class c Where c.#name='PublicUniversity'), (Select #its.RightPart.Prop.#oid from #P.Right.Part Where #its.RightPart.Prop.#name='Name'), [Select #oid From #Propert p Where p.#name='IdUniv'])

![Fig. 9. Exploitation of PFD in generating a normalized logical model.](image)

**Step 2.** To identify canonical and non-canonical classes, we use the algorithm described in Section 3.1. Note that the second solution described in figure 6 is chosen.
Step 3. We exploit PFD defined on each CC to generate normalized relations per CC. Figure 9 shows an example of the generation of the normalized logical model of PublicUniversity class. Normalized tables are stored in the data part of OntoDB.

Step 4. To place non canonical classes in the ontology hierarchy of OntoDB ontology part, we use Pellet 1.5.2 [6]. Figure 10 shows an example of the asserted and inferred hierarchy of University, PublicUniversity and PrivateUniversity classes.

Fig. 10. Use of Pellet in generating classes subsumption relationship

5 Conclusion

This paper presented a five steps methodology handling the consistent design of an OBDB from the conceptual model till the logical model. The approach considers local ontologies as conceptual models and borrows formal techniques issued from both graph theory for dependency analysis, from description logics reasoning for class placement and from relational database theory for creating relational views, on normalized tables, associated to canonical classes. To the best of our knowledge, this work is the sole that considers different types of dependencies between ontological concepts at different design levels of an integrated methodology. This approach is sound and consistent; the different refinement steps preserve the functionalities offered by the local ontology. It is based on formal models and is independent of the chosen target OBDB architecture. Finally, this approach is tool supported and some experiments were conducted within the produced tool suite.

This work led several open research issues. Among them, we can cite 1) need of validation through a wider set of case studies and deployment in an industrial setting, 2) handling evolution and instance migration in case the ontology evolves
or in the presence of integration requirements and 3) proposition of optimization structures (e.g., indexing schemes) during the physical design phase.

References


