triples or use specific storage approaches on these instances [3–5]. Languages exploiting such models have been proposed. Examples are RQL [6], OWL-QL [7] etc. They support several specific features like instance resonating, graph traversal etc. However, they do not preserve any database compatibility with database models. Therefore, migration of instances is necessary.

If several persistence data models and query languages have been proposed by artificial intelligence community for the semantic web, few work have been conducted and originated from a database-oriented perspective. In this paper, we present *OntoQL*, an exploitation language for an OBDB data model, named OntoDB designed by a layered approach on top of a relational database model. It has been proved useful for several database applications (e.g. semantic integration [8]) in several application domains (electronic, automotive, medical data).

This paper is structured as follows. Next section presents the formalization of the OBDB data model addressed in this paper. Section 3 presents the algebra designed for the *OntoQL* query language and section 4 discusses optimization techniques for this algebra. Section 5 introduces the *OntoQL* data definition and query languages by a set of examples showing the differences with traditional query languages. Section 6 discusses the *OntoQL* implementation and processing issues when implemented on top of an object-relational database (ORDBMS). Section 7 describes, on a toy example, how our approach runs. Section 8 discusses related work. Finally, section 9 concludes the paper by summarizing the main results and suggesting future work.

2 The OBDB Data Model

The OBDB database model is based on the definition of two main related parts: *ontology* and *content*. Instances are stored in the content part while ontologies are stored in the ontology part.

2.1 Ontology.

The ontology part allows to store ontologies as instances of an ontology model. It is formally defined by a 7-Tuple as $<$ `E`, `OC`, `A`, `SuperEntities`, `TypeOf`, `AttDomain`, `AttRange`, `Val` $>$, where:

- `E` is a set of entities representing the ontology model. It provides with a global super entity `Concept`, the predefined entities `C` and `P` described below and user-defined entities.
- `OC` is the set of concepts of ontologies (classes, properties ...) available in the database or that can be constructed by a query. All the concepts of ontologies have an unique identifier.
- `A` is the set of attributes describing each ontology concept.

- $\texttt{SuperEntities} : \texttt{E} \rightarrow 2^{\texttt{E}}$[1] is a partial function associating a set of super entities to an entity. It defines a lattice of entities. Its semantics is inheritance and it ensures substitutability.
- $\texttt{TypeOf} : \texttt{OC} \rightarrow \texttt{E}$ associates to each concept of an ontology the lower (strongest) entity in the hierarchy it belongs to.
- $\texttt{AttributeDomain}, \texttt{AttributeRange} : \texttt{A} \rightarrow \texttt{E}$ define respectively the domain and the range of each attribute.
- $\texttt{Val} : \texttt{OC} \times \texttt{A} \rightarrow \texttt{OC}$ gives the value of an attribute of an ontology concept.

The OBDB data model provides with atomic types (`Int`, `String`, `Boolean`) and with two parameterized types `Set`[T] and `Tuple`. `Set`[T] denotes a type for collections of elements of type T and $\{\texttt{o}_1, \ldots, \texttt{o}_\texttt{n}\}$ is an object of this type (the $\texttt{o}_\texttt{i}$'s are objects of type T). The $\texttt{Tuple}[< (\texttt{A}_1, \texttt{T}_1), \ldots, (\texttt{A}_\texttt{n}, \texttt{T}_\texttt{n}) >]$ parameterized type creates relationships between objects. It is constructed by providing a set of attribute names ($\texttt{A}_\texttt{i}$) and attribute types ($\texttt{T}_\texttt{i}$). $\texttt{Tuple}[< (\texttt{A}_1, \texttt{T}_1), \ldots, (\texttt{A}_\texttt{n}, \texttt{T}_\texttt{n}) >]$ denotes a type tuple constructed using the $\texttt{A}_\texttt{i}$ attribute names and $\texttt{T}_\texttt{i}$ attribute types. $< \texttt{A}_1 : \texttt{o}_1, \ldots, \texttt{A}_\texttt{n} : \texttt{o}_\texttt{n} >$ is an object of this type (the $\texttt{o}_\texttt{i}$'s are objects of type $\texttt{T}_\texttt{i}$). The `Tuple` type is equipped with the $\texttt{Get\_A}_\texttt{i}\texttt{\_value}$ functions to retrieve the value of a `Tuple` object `o` for the attribute $\texttt{A}_\texttt{i}$. The application of this function may be abbreviated using the dot-notation ($\texttt{o.A}_\texttt{i}$)

`E` provides the predefined entities `C` and `P`. Instances of `C` and `P` are respectively the classes and properties of the ontologies. Entity `C` defines the attribute $\texttt{SuperClasses} : \texttt{C} \rightarrow \texttt{SET}[\texttt{C}]$ and entity `P` defines the attributes $\texttt{PropDomain} : \texttt{P} \rightarrow \texttt{C}$ and $\texttt{PropRange} : \texttt{P} \rightarrow \texttt{C}$. The description of these attributes is similar to the definitions given for `SuperEntities`, `AttributeDomain` and `AttributeRange` replacing entities by classes and attributes by properties. Moreover, a global super class `Root` is predefined.

Finally, an ontology gives a precise definition of concepts with more attributes (comment, version, multi-lingual definition, synonymous names, ...) to describe classes and properties of ontologies. These predefined entities and attributes constitute the kernel of the ontology models we have considered. User-defined entities (illustration, document, ...) and attributes (note, remark, ...) may be added to this kernel in order to take into account other specific ontology models.

Notice that the ontologies stored in this part are defined with the different characteristics shared by the standard ontology models PLIB[9] and OWL[2]. The *multi-instantiation* and *property subsumption* (subproperty) specific features of OWL are not taken into account in this formalization. They have been removed to get an optimal database representation (OBDB Light). However, a full version of the OBDB model with these features is under development (OBDB Full). Nevertheless, for several examples and significant applications[2] [3] we have developed, this limitation had neither effect nor impact on the expressive power of the treated problems. [10] gives our precise view of ontologies with respect to their exploitation in a database context.

[1] We use the symbol $2^{\texttt{E}}$ to denote the power set of E.

[2] See http://www.plib.ensma.fr for references to IEC/ISO standards ontologies

[3] For example, the EPICEM project (http://www.episem-action.org)

2.2 Content.

The content part stores instances of ontology classes. It is formalized by a 5-tuple $< \texttt{EXTENT}, \texttt{I}, \texttt{TypeOf}, \texttt{SchemaProp}, \texttt{Val} >$ where:

- $\texttt{EXTENT}$ is a set of *extensional definitions* of ontology classes.
- $\texttt{I}$ is the set of instances of the OBDB. Each instance has an identity.
- $\texttt{TypeOf} : \texttt{I} \rightarrow \texttt{EXTENT}$ associates to each instance the extensional definition of the class it belongs to (collection of its instances).
- $\texttt{SchemaProp} : \texttt{EXTENT} \rightarrow 2^{\texttt{P}}$ gives the properties used to describe the instances of an extent (the set of properties valued for its instances).
- $\texttt{Val} : \texttt{I} \times \texttt{P} \rightarrow \texttt{I}$ gives the value of a property occurring in a given instance. This property must be used in the extensional definition of the class the instance belongs to.

2.3 Relationship Between each Part.

The relationship between ontology and its instances (content) is defined by the partial function $\texttt{Nomination} : \texttt{C} \rightarrow \texttt{EXTENT}$. It associates a definition by intension with a definition by extension of a class. Classes without extensional definition are said to be *abstract*. The set of properties used in an extensional definition of a class must be a subset of the properties defined in the intensional definition of a class ($\texttt{propDomain}^{-1}(\texttt{c}) \supseteq \texttt{SchemaProp}(\texttt{nomination}(\texttt{c}))$).

2.4 Related work.

Storing ontologies and their instances in databases has been the subject of several research studies and proposals. In the context of the semantic web, several OBDB models [3–5, 11] have been proposed to manage data described by ontologies represented in the standard ontology models RDFS [1] or OWL [2]. In these approaches, an instance, often called an individual, has its own property and class structure. Therefore, to manage instances, a generic database schema, not meaningful to an user and not customizable, is used. The simplest and more general one uses an unique table of triples [11] for storing both the ontology and its instances. Other approaches [3, 4] separate the representation of the ontology and its instances in two parts. In these approaches, the most common practice for storing instance data is to use the so-called vertical model [12] where information is stored in triples `(subject, property, value)` a variant of which, called the binary model is to have one table per property that contains only pairs of the form `(subject, value)`.

Our approach differs from the ones listed previously. Indeed, the conceptual model of the instances is part of the OBDB data model (`EXTENT`, `SchemaProp` in the formalization of this data model). This conceptual model may be created and customized by users from the ontology (see section 5.1). Thus, many different logical models may be derrived/related to the same ontology. This possibility promotes a database approach preserving compatibility with RDBMs and promotes semantic integration of OBDBs by offering an ontology for different logical models.

3 Query Algebra for OBDB

The specific aspects of our OBDB data model, where not all the values of properties of a class are required in an instance of this class, raised the necessity to define an exploitation language for such OBDBs. [13] provides with the precise requirements we have set up at the beginning of our work for designing such a language. More details about the positioning of this language among the different database models are given in section 8. To reach this goal, we suggest to build an algebra *OntoAlgebra*, for managing OBDB databases.

Since the OBDB model uses extensively object-oriented database (OODB) features, we suggest to specialize, extend and reuse the operators issued from the *ENCORE* algebra [14] in order to get benefits of the work achieved in the context of OODBs. Signatures of the operators defined on the OBDB data model belong to $(\mathtt{E} \cup \mathtt{C}) \times 2^{\mathtt{OC} \cup \mathtt{I}} \rightarrow (\mathtt{E} \cup \mathtt{C}) \times 2^{\mathtt{OC} \cup \mathtt{I}}$. These main operators of this algebra are *OntoImage*, *OntoProject*, *OntoSelect* and *OntoOJoin*. For clarity purpose, solely these operators are formally presented below restricted to the signature $\mathtt{C} \times 2^{\mathtt{I}} \rightarrow \mathtt{C} \times 2^{\mathtt{I}}$. However, the defined semantics is adapted for querying both ontology, content and simultaneously ontology and content parts.

- OntoImage. The *OntoImage* operator returns the collection of objects resulting from applying a function to a collection of objects. Its signature is $\mathtt{C} \times 2^{\mathtt{I}} \times \mathtt{Function} \rightarrow \mathtt{C} \times 2^{\mathtt{I}}$. `Function` contains all the properties in `P` and all properties that can be defined by composing properties of `P` (path expressions). Differently from the object-oriented data model, the OBDB data model authorizes the fact that one or more of the properties occurring in the function parameter may not be valued in the extensional definition of the class. Notice that this capability weakens the data model in order to support richer and flexible descriptions than those allowed in classical OODBs. Thus, it becomes necessary to extend the domain of the `Val` function to the properties defined on the intensional definition of a class but not valued on its extensional definition. This extension requires the introduction of the `UNKNOWN` value. We call `OntoVal` this extension of `Val`. It is defined by:

$$\mathtt{OntoVal}(\mathtt{i},\mathtt{p}) = \mathtt{Val}(\mathtt{i},\mathtt{p}), \text{ if } \mathtt{p} \in \mathtt{SchemaProp}(\mathtt{TypeOf}(\mathtt{i})) \text{ else, } \mathtt{UNKNOWN} \ .$$

`UNKNOWN` is a special instance like `NULL` is a special value for SQL. Whereas `NULL` may have many different interpretations like value unknown, value inapplicable or value withheld, the only interpretation of `UNKNOWN` is value unknown, i.e., there is some value, but we don't know what it is. To preserve composition, `OntoVal` applied to a property which value is `UNKNOWN` returns `UNKNOWN` (strict interpretation). So, the semantics of `OntoImage` is defined by:

$$\mathtt{OntoImage}(\mathtt{T}, \{\mathtt{i}_1, \ldots, \mathtt{i}_n\}, \mathtt{f}) = (\mathtt{PropRange}(\mathtt{f}), \{\mathtt{OntoVal}(\mathtt{i}_1, \mathtt{f}), \ldots, \mathtt{OntoVal}(\mathtt{i}_n, \mathtt{f})\}) \ .$$

- OntoProject. The *OntoProject* operator extends *OntoImage* allowing the application of more than one function to an object. The result type is a `Tuple`

which attribute names are taken as parameter. It is defined by:

$$\begin{aligned}&\texttt{Project}(\texttt{T}, \texttt{I}_\texttt{t}, \{(\texttt{A}_1, \texttt{f}_1), \ldots (\texttt{A}_\texttt{n}, \texttt{f}_\texttt{n})\}) = \\ &\quad (\texttt{Tuple}[< (\texttt{A}_1, \texttt{PropRange}(\texttt{f}_1)), \ldots, (\texttt{A}_\texttt{n}, \texttt{PropRange}(\texttt{f}_\texttt{n})) >], \\ &\qquad \{< \texttt{A}_1 : \texttt{OntoVal}(\texttt{i}, \texttt{f}_1), \ldots, \texttt{A}_\texttt{n} : \texttt{OntoVal}(\texttt{i}, \texttt{f}_\texttt{n}) > | \texttt{i} \in \texttt{I}_\texttt{t}\}) \ .\end{aligned}$$

It returns the type of elements together with the set of corresponding values.
- OntoSelect. It creates a collection of objects satisfying a selection predicate. Its signature is $\texttt{C} \times 2^\texttt{I} \times \texttt{Predicate} \rightarrow \texttt{C} \times 2^\texttt{I}$ and its semantics is defined by:

$$\texttt{OntoSelect}(\texttt{T}, \texttt{I}_\texttt{t}, \texttt{pred}) = (\texttt{T}, \{\texttt{i} | \texttt{i} \in \texttt{I}_\texttt{t} \wedge \texttt{pred}(\texttt{i})\}) \ .$$

If the predicate taken as parameter of *OntoSelect* contains function applications, then `OntoVal` must be used. So, operations involving `UNKNOWN`, that may appear in a predicate, must be extended to handle this value (interpreted like `NULL`). If any operator involves this value as parameter, then it returns `UNKNOWN` (strict interpretation).
- OntoOJoin. It creates relationships between objects of two collections.

$$\begin{aligned}&\texttt{OntoOJoin}(\texttt{T}, \texttt{I}_\texttt{t}, \texttt{R}, \texttt{I}_\texttt{r}, \texttt{A}_1, \texttt{A}_2, \texttt{pred}) = \\ &(\texttt{Tuple}[< (\texttt{A}_1, \texttt{T}), (\texttt{A}_2, \texttt{R}) >], \{< \texttt{A}_1 : \texttt{t}, \texttt{A}_2 : \texttt{r} > | \texttt{t} \in \texttt{I}_\texttt{t} \wedge \texttt{r} \in \texttt{I}_\texttt{r} \wedge \texttt{pred}(\texttt{t}, \texttt{r})\}) \ .\end{aligned}$$

- Operator *. It is the explicit polymorphic operator to distinguish between queries on instances of a single class `C` and instances of all the classes *subsumed* by `C` and denoted $\texttt{C}^*$. $\texttt{ext} : \texttt{C} \rightarrow 2^\texttt{I}$ returns the instances of a class and $\texttt{ext}^* : \texttt{C} \rightarrow 2^\texttt{I}$ its deep extent. If `c` is a class and $\texttt{c}_1, \ldots \texttt{c}_\texttt{n}$ are the direct sub-classes (in the sense of the subsumption relationship) of `c`, `ext` and $\texttt{ext}^*$ are derived recursively[4] by:

$$\begin{aligned}&\texttt{ext}(\texttt{c}) = \texttt{TypeOf}^{-1}(\texttt{Nomination}(\texttt{c})) \ . \\ &\texttt{ext}^*(\texttt{c}) = \texttt{ext}(\texttt{c}) \cup \texttt{ext}^*(\texttt{c}_1) \cup \ldots \cup \texttt{ext}^*(\texttt{c}_\texttt{n}) \ .\end{aligned}$$

The `ext` and $\texttt{ext}^*$ make it possible to define the * operator as $^* : \texttt{C} \rightarrow \texttt{C} \times 2^\texttt{I}$ where$^*(\texttt{T}) = (\texttt{T}, \texttt{ext}^*(\texttt{T}))$.

In addition to these main operators, *OntoAlgebra* includes set operations (*OntoUnion*, *OntoDifference*, and *OntoIntersection*) and collection operations (*OntoFlatten*, *OntoNest* and *OntoUnNest*).

Next section shows how operators of *OntoAlgebra* can be optimized using the characteristics of the OBDB data model.

4 Optimizations of *OntoAlgebra*'s Operators

As identified when defining the OBDB model, some of the properties occurring in the ontology part may not be valued (and thus not available) in the corresponding instances of the content part. This is a main difference with OODBs

[4] To simplify notation, we extend all functions `f` by $\texttt{f}(\emptyset) = \emptyset$

where properties of a class are valued in the instances. As a consequence, important optimizations based on partial evaluation techniques can be set up. Indeed, it is not necessary to search the values of a property in the instances of a class using a property defined as not available in the extensional definition of this class. This source of optimizations is characterized by the formal logical expression(invariant property) (1). When this expression evaluates to true, it becomes possible to reduce, and sometimes avoid, accesses and traversals of the content part which is cost effective.

Let $\mathtt{p} \in \mathtt{P}$, let $\mathtt{C}$ be a class, let $\mathtt{I_c}$ be a set of instances of $\mathtt{C}$,

$$\mathtt{p} \notin \mathtt{SchemaProp}(\mathtt{Nomination}(\mathtt{C})) \Rightarrow \mathtt{OntoImage}(\mathtt{C}, \mathtt{I_c}, \mathtt{p}) = \{\mathtt{UNKNOWN} | \mathtt{i} \in \mathtt{I_c}\} \ . \quad (1)$$

This optimization can be generalized to the *OntoProject* operator. It can also be applied to the *OntoSelect* and *OntoOJoin* operators when the predicate taken in parameter of these operators involves the use of a property. More precisely, assume this predicate is in conjunctive normal form and that one of the conjunctive element involves a property satisfying the logical expression (1), then, the *OntoSelect* and *OntoOJoin* operators return the empty set.

This optimization is only available for local queries on instances of a class $\mathtt{C}$. The operator * and path expressions introduce polymorphism. To optimize the polymorphic operators of the *OntoAlgebra*, it is necessary to translate them into non polymorphic operators acting on the class at each level of the polymorphic hierarchy (flattening the hierarchy).

Assume $\mathtt{p_1}$ and $\mathtt{p_2}$ are two properties which domains are respectively $\mathtt{C_1}$ and $\mathtt{C_2}$. Path expressions involving these two properties can be decomposed according to the following algebraic law:

$$\mathtt{OntoImage}(\mathtt{C_1}, \mathtt{ext}(\mathtt{C_1}), \mathtt{p_1} \circ \mathtt{p_2}) \Leftrightarrow \mathtt{OntoImage}(\mathtt{LeftOuterOntoOJoin}(\mathtt{C_1}, \mathtt{ext}(\mathtt{C_1}),^* (\mathtt{C_2}), \mathtt{A_1}, \mathtt{A_2}, \mathtt{A_1.p_1} = \mathtt{A_2.oid}), \mathtt{A_2.p_2}) \quad (2)$$

Since a result is required for each instance of $\mathtt{C_1}$, a *left outer join* is necessary. This decomposition is easily extended to paths of any length. Moreover, the same equivalence may be used for the *OntoProject* operator by decomposing each path expression taken in parameter in the same way.

Path expressions may also appear in the predicate of the *OntoSelect* and *OntoOJoin* operators. For example, assume the predicate is $\mathtt{p_1} \circ \mathtt{p_2} \ \theta \ \mathtt{c}$ where θ is a comparison operator and $\mathtt{c}$ a constant, then, the following equivalence can be used to decompose this predicate :

$$\begin{aligned}\mathtt{OntoSelect}(\mathtt{C_1}, \mathtt{ext}(\mathtt{C_1}), \mathtt{p_1} \circ \mathtt{p_2} \ \theta \ \mathtt{c}) \Leftrightarrow \mathtt{OntoImage}(\mathtt{OntoSelect}(\\ \mathtt{LeftOuterOntoOJoin}(\mathtt{C_1}, \mathtt{ext}(\mathtt{C_1}),^* (\mathtt{C_2}), \mathtt{A_1}, \mathtt{A_2}, \mathtt{A_1.p_1} = \mathtt{A_2.oid}), \\ \mathtt{A_2.p_2} \ \theta \ \mathtt{c}), \mathtt{get\_A_1\_Value}) \ . \end{aligned} \quad (3)$$

The * operator can be removed using the set operator *OntoUnion*. Assume $\theta_1 \ldots \theta_\mathtt{n}$ to be a set of *OntoAlgebra* operators, $\mathtt{C}$ a class having $\mathtt{C_1} \ldots \mathtt{C_n}$ as direct sub-classes. Removing * operator is preformed by applying the following

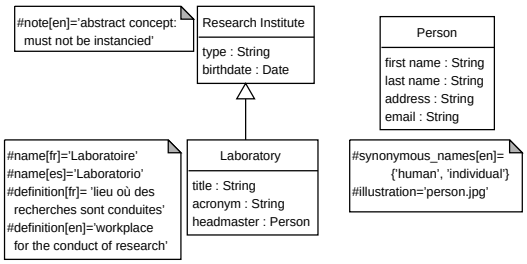

Fig. 1. Ontology example

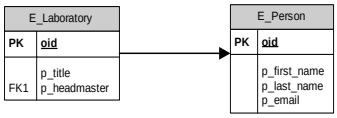

Fig. 2. Content example

equivalence recursively (unfolding operation).

$$\theta_1(\ldots\theta_n(^*(\mathtt{C})) \Leftrightarrow \theta_1(\ldots\theta_n(\mathtt{C},\mathtt{ext}(\mathtt{C}))\ \mathtt{OntoUnion}$$
$$\theta_1(\ldots\theta_n(^*(\mathtt{C_1}))\ \mathtt{OntoUnion}\ldots\mathtt{OntoUnion}\ \theta_1(\ldots\theta_n(^*(\mathtt{C_n})) \quad (4)$$

The (1), (2), (3) and (4) algebraic laws will be exploited by the query plans presented in section 6 below.

5 The OntoQL Language

OntoQL is the OBDB exploitation language built on the defined database model. An overview of the querying capabilities of OntoQL has been given in [15]. Its semantics is given by the *OntoAlgebra* previously defined. This section presents the details of this language (DDL and QL). A toy example, presented on figure 1, is used along this description in order to avoid complex syntactic definitions. Next subsections focus on the specific parts of this language. We will discuss the positioning of this language among database and semantic web languages in section 8.

5.1 The Data Definition (DDL) and Manipulation (DML) Parts of OntoQL

OntoQL allows to create, alter and drop concepts of ontologies (classes, properties ...) as well as their attributes values (name, definition ...). Let's consider the following expression :

```
CREATE #CLASS Laboratory EXTENDS "Research Institute" (
 DESCRIPTOR(#name[fr,es] = ('Laboratoire','Laboratorio'),
            #definition[fr] = 'lieu où des recherches sont conduites',
            #definition[en] = 'workplace for the conduct of research')
 PROPERTIES(title String, headmaster Person, acronym String) );
```

This expression creates an ontology class named `Laboratory` in English (the default language) as a subclass of `Research Institute`. Thus, it inherits the properties of `Research Institute`. `DESCRIPTOR` and `PROPERTIES` clauses make the distinction between the definition of the attributes values describing a class (name,

remark, note ...) and the definition of its characterisation properties (those fixed by the user). This distinction is carried by the attributes prefix `#` (see section 5.2).

On the content part, an extent can be attached to this class by the following expression:

```
CREATE EXTENT OF Laboratory (title, headmaster);
```

Notice that the `acronym` property will not be valued in the content part. One may define another content for the class `Laboratory` according to another logical database model. When executed, this expression creates a relational schema presented in figure 2 to store instances of this class.

Data Manipulation Language (DML) operators are also provided by OntoQL. These operators may add, delete and update each parts of an OBDB. Indeed, concepts and their instances may be managed by these operators. Moreover, since the metadata describing the ontology model are themselves stored in a relational database, the same operators allow adding/deleting/updating attributes of the ontology model (e.g. adding the attributes `unit`, `comment`, etc. defining a new translation language more than English, etc.).

5.2 The Query Language Part of OntoQL

The query language part of *OntoQL* is designed as an extension of SQL to query ontologies, their contents and both ontologies and contents stored in an OBDB. The syntax of a query is given by:

```
  SELECT attributeList, propertyList, iteratorList
    FROM iteratorDeclarationList
   WHERE condition
GROUP BY attributeList, propertyList, iteratorList
  HAVING condition
ORDER BY attributeList, propertyList
```

where `attributeList` (resp. `propertyList`) is a list of attributes (resp. properties); `iteratorList` is a list of iterators declared in the `FROM` clause;
`iteratorDeclarationList` introduces iterators over a set of entity and/or class instances. Moreover, the `SELECT` block of *OntoQL* supports the following features, all expressed by *OntoAlgebra* operators composition.

- *Path expressions.* Associations may be traversed using the dot notation.
- *Polymorphic query.* The `*` operator is provided to distinguish between local queries on instances of a class/entity `C` and instances of all the classes/entities denoted `C*` subsumed by `C`.
- *Dependent collection.* A collection returned as the value of a property/attribute may be traversed using an iterator introduced in the `FROM` clause.
- *Nested queries.* Queries may be nested not only in the `WHERE` clause but also in the `SELECT` and `FROM` clauses.
- *Aggregate functions. OntoQL* provides aggregate functions `count`, `sum`, `avg`, `min` and `max`.

– *Quantification.* Existential (`ANY`, `SOME`) and universal (`ALL`) quantification may be expressed.
– *Set operators.* `Union`, `Intersection` and `Except` operators are provided.

Next subsections show how this general model of an *OntoQL* query is used to express query on ontology, on content and both on ontology and on content. Moreover, these sections show on several query examples, specific usages of *OntoQL* to exploit ontology characteristics, content characteristics or both.

Ontology Querying. Ontologies querying retrieve descriptive information from the ontology part. The `FROM` clause of an ontology query introduces iterators over instances of predefined entities (`class`, `property`) of the ontology model as well as on user-defined entities. The `SELECT` clause defines projection on predefined attributes (`name`, `definition`, `scope`, `superclasses` ...) and user-defined attributes. The value of some attributes, such as `name` are given in different natural languages. The query `Q1` searches for the English name of the class which French name is "`Institut de Recherche`".

```
Q1. SELECT #name[EN] FROM #class WHERE #name[FR]='Institut de Recherche'
```

Remember that the prefix `#` is used to distinguish between attributes of entities and properties of classes.

Moreover classes and properties are implicitly named. For example, the query `Q2` uses `Research Institute` class name to retrieve the name in French of the properties defined on this class.

```
Q2.SELECT p.#name[FR] FROM "Research Institute".#properties
```

This capability is also offered by OQL. However, to use it on a given object of a class, users must explicitly name this object.

Content Querying. The `FROM` clause of a content query introduces iterators over instances of ontology classes and the `SELECT` clause defines projection on properties defined on this class but not necessary provided by the `CREATE EXTENT` clause. The following queries search for the names of all laboratories with an English (`Q3a`) a French (`Q3b`), using external identifiers (`Q3c`) and internal identifiers (`Q3d`) queries:

```
Q3a.SELECT acronym   FROM Laboratory  Q3c.SELECT @710C-01 FROM @7194-01
Q3b.SELECT accronyme FROM Laboratoire Q3d.SELECT !1012 FROM !1068
```

Here `!x` and `@x` are respectively internal identifiers (known by database developers) and external references (like URI).

Ontology and Content Querying. *OntoQL* introduces an iterator over instances of classes retrieved by an ontology query. `Q4a` query illustrates this feature using a *dependant collection.*

```
Q4a.SELECT i.oid, i.p, p.#name[en]
      FROM C in #class, p in C.#properties, i in C*
     WHERE C.#name[fr] like 'Per%'
```

`Q4b` is equivalent to `Q4a`. It uses a `SELECT` operator in the `FROM` clause to access simultaneously the ontology and the content parts (*nested query*).

```
Q4b.SELECT i.oid, i.p, p.#name[en]
      FROM C in (SELECT C FROM C in #class
                  WHERE C.#name[fr] like 'Per%'),
           p in C.#properties, i in C*
```

This query retrieves classes which name begin with `"Per"`. The iterator `i` ranges over instances of these classes and is used to access and return property English name and value for these instances.

OntoQL proposes the operator `typeof` to retrieve the base class of a content instance. This operator allows to express queries from the content to the ontology. For example, the query `Q5` searches for the address and email of all polymorphic instances of the class `Person` and uses the operator `typeof` to retrieve the French name of the base class of these instances.

```
Q5.SELECT i.address, i.email, typeof(i).#name[fr] FROM i in Person*
```

The `typeof` operator returns only one base class for an instance. This query could not be written this way if multi-instantiation was allowed.

6 Processing of OntoQL

We have implemented *OntoQL* and the *OntoAlgebra* operators on an OntoDB prototype. Demonstrations of this prototype are available at http://www.plib.ensma.fr/plib/demos/ontodb/index.html. This section briefly outlines how these operators are processed on this platform.

The prototype considered is an implementation of the OntoDB data model in a object-relational database (ORDBMS), namely PostgreSQL [16]. In this prototype, the link between the ontology and its content parts are defined using an identifier. To simplify, assume that the identifier of the ontological (intensional) definition of a class is `cid`, then its content (extensional) definition is represented by a table identified by `ecid`. A similar mechanism for properties is used. Indeed, assume that the identifier of a property is `pid` in the intensional definition of a class, then it is represented by a column identified by `ppid` if used in the content definition.

To process *OntoAlgebra* operators, they are translated in the underlying query language, i.e. SQL99. With this approach, the optimization process is split into two sub-processes. The first one is related to the *OntoQL* engine and the second one is performed by the underlying database engine. The translation process follows six identified steps.

1. **Logical query plan generation.** The query, written in *OntoQL*, is parsed and turned into an expression tree involving *OntoAlgebra* operators in its nodes (logical query plan). Entities, classes, properties and attributes occur in leaves.
2. **Logical query plan transformation.** Path expressions and * operators are removed from the logical query plan using equivalence algebraic laws (2), (3) and (4) defined on *OntoAlgebra* (see section 4). In this step, we use an algorithm avoiding multiple decomposition of identical paths and thus avoiding unnecessary join operations.
3. **Optimize the tree.** The optimization situations, identified in section 4, are used to reduce the logical query plan. This step is performed together with the previous step to avoid duplicating unnecessary parts of the tree.
4. **Translation of an *OntoAlgebra* tree to relational algebra trees.** This translation is achieved by applying the following rules:
   (a) the identifier of the intensional definition of a class is replaced by the identifier of its extensional definition. If the class is abstract then its identifier is replaced by the name of an empty table (e.g., Dual in Oracle);
   (b) if the property is not used in the extensional definition of a class, `UNKNOWN` is translated to `NULL`.
   (c) *OntoImage* and *OntoProject* are translated into the projection operator of the relational algebra. Other operators of *OntoAlgebra* are translated into their relational counterpart.

   An *OntoQL* query often requires access to the content of an OBDB according to the query on the ontology part. Thus, this translation may require to build more than one relational algebra tree.
5. **Optimization of the relational algebra trees.** This step consists in using the different algebraic laws that hold for relational algebra to turn the relational trees into equivalent trees that may be executed more efficiently by the underlying ORDBMS. The ORDBMS optimizer may perform other optimizations it supports.
6. **Translation of the relational algebra trees into SQL queries.** The optimized relational trees are translated into SQL queries according to the underlying ORDBMS and executed to get the *OntoQL* query result.

7 Example

To illustrate our language proposal, let us develop a practical example showing how a query, written in *OntoQL*, is processed. This example extends the previous one by precising the address property in the class person.

An Example of Data Model. Figure 3 shows an UML data model. Specific annotations `d`/`v` are added to the property names to take into account the specific features of the OBDB data model. `d` means that this property is defined on this class ; `v` means that this property is valued in the extensional definition of this class. This schema is defined to manage names of persons. Students and

employees are also described by their addresses. Since `Address` is an abstract class (its name is in italic), addresses are located either in the USA or in France. Notice that for French addresses, the property `state` is not valued on instances.

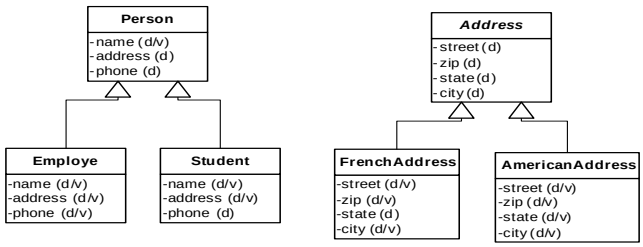

Fig. 3. Schema example

An Example of Query. Assume we want to find in which cities of the state Utah some persons are living. To answer this query, an *OntoQL* statement using English attributes (left) or French ones (right) may be written:

```
SELECT address.city              SELECT adresse.ville
  FROM Person*             <=>     FROM Personne*
 WHERE address.state='Utah'       WHERE adresse.etat='Utah'
```

Query Processing Steps. The first step of our processing generates the logical plan presented in Fig. 4 (a). The path expressions are removed from this query plan. If the application of the property `address` is decomposed in each path, the transformed logical plan is the one presented in Fig. 4 (b). In this logical plan, the upper left outer join is unnecessary. Therefore, application of the property `address` is decomposed only once and paths composed with this property are changed using an alias of the class `Address`. The query plan resulting from this processing is presented in Fig. 4 (c). In step 3 and 4, * operators applied to `Person` and `Address` are removed. Because `address` is not used on classes `Person`, optimization (1) allows to deduce that the result of the *LeftOuterOntoOJoin* is $\{< \mathtt{p}, \mathtt{UNKNOWN} > | \mathtt{p} \in \mathtt{ext(Person)}\}$. Thus, the predicate $\mathtt{a.state} =' \mathtt{Utah}'$ is always `UNKNOWN`. As a consequence, this query doesn't return any result for the class `Person`. Therefore our logical query plan must be duplicated for the classes `Employee` and `Student` only. Let's consider removing of * from class `Address`. Because the property `state` is not used on classes `Address` and `FrenchAddress` the result of the *OntoSelect* operator is empty and it is not necessary to run it. Finally, our logical query plan must only be duplicated for the `AmericanAddress` class. Logical query resulting from this processing is presented in Fig. 4 (d).

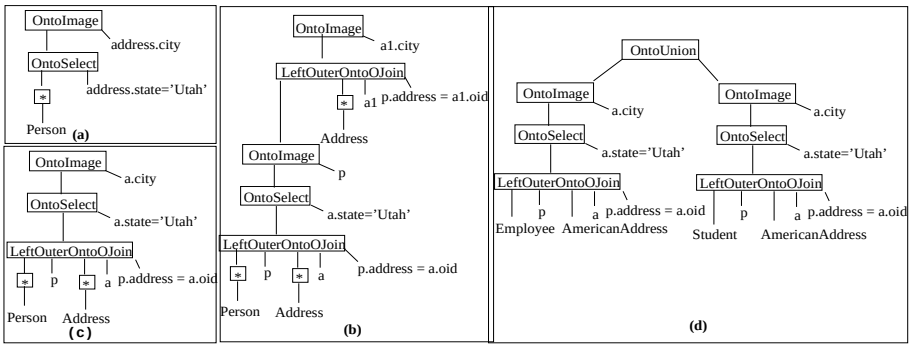

Fig. 4. Logical query plans for our query example.

In this example, translation of this query plan to a relational query plan is straightforward. It consists, first, in replacing names of classes and properties by names of tables and columns corresponding to their extensional definitions, and, second, in switching *OntoAlgebra* operators to their corresponding relational operators. In step 6, the relational tree may be modified to replace operators not supported by the underlying DBMS (e.g., *outer join*) or to optimize it according to the generation and optimization of the logical query plan supported by the DBMS. Without modification of our logical query plan, the obtained SQL query running on the underlying DBMS is:

```
SELECT a.pcity
  FROM eEmployee p LEFT OUTER JOIN eAmericanAddress a
                                ON p.paddress = a.oid
 WHERE a.pstate='Utah'
UNION
SELECT a.pcity
  FROM eStudent s LEFT OUTER JOIN eAmericanAddress a
                                ON s.paddress = a.oid
WHERE a.pstate='Utah'
```

This query returns the results of our *OntoQL* query.

8 Related Work

OntoQL is a language based on a database model for exploiting ontologies and the knowledge they describe. Therefore, one can compare it with database languages on the one hand and with semantic web based languages on the other hand.

8.1 Database Exploitation Languages

Compared to classical database languages, *OntoQL* preserves upward compatibility with existing database exploitation languages associated to different layered database models.

- RDB. When an user is aware of internal identifiers for tables and columns of the OBDB model, classical SQL can be used to retrieve and manage table data.
- ORDB. When all properties values of a class are provided at the instance level, one can use the OBDB model as an OODB model and use *OntoQL* constructs as an exploitation language.
- When the previous conditions are not fulfilled, we can use *OntoQL* as an ontology exploitation language as shown in this paper.
- Finally, the top layer is the one of a linguistic based exploitation of an OBDB. Indeed, when the attributes #name, #remark, #synonymous, #translations are exploited by *OntoQL* constructs, it is possible to envisage a linguistic exploitation as shown in queries Q2 and Q3 given in section 5.2 .

Moreover,*OntoQL* has been defined as an extension of SQL to exploit an OBDB model defined for exploiting data and their semantics and for semantic integration. According to these applications, related languages are multidatabase languages like SchemaSQL [17] or MSQL [18]. *OntoQL* shares with these languages the capability to express queries on data independently of their schemas. However, whereas these languages use the system catalog as an abstraction for database schemas, *OntoQL* uses the ontologies themselves to encode this abstraction providing a dynamic approach for encoding different abstractions corresponding to different point of views of application domains. Consequently, *OntoQL* presents many differences with these languages such as its object-oriented nature or its independency w.r.t the model (relational, object-relational, object) used to represent the schemas of the data.

The SOQA-QL [19] language (SIRUP project) allows querying ontologies and the data they describe independently of the ontology model and of the hardware/software used platform. Like *OntoQL*, the main application of SOQA-QL is semantic integration. Moreover, they are both based on SQL and defined on a core ontology model (the SOQA Ontology Meta Model for SOQA-QL) representing the shared modelling capabilities of some ontology models in order to provide an access to data independently of the used ontology model. Nevertheless, there are crucial differences between these two languages. First, in the opposite of SOQA-QL, the core ontology model of *OntoQL* can be extended to take into account particularities of some ontology models (e.g. adding new attributes that characterize ontology model concepts). To provide this capability, *OntoQL* is based on an algebra not tight to the core ontology model whereas the SOQA-QL algebra (i.e, encoded in the SIRUP Ontology Query API) provides access methods for all ontological components defined in the SOQA Ontology Meta Model and the user does not have the possibility to dynamically update this API. Another difference is that SOQA-QL and OntoQL do not keep the

same level of compatibility with SQL. Indeed, whereas SOQA-QL queries on ontologies are expressed in a SQL-like syntax, SOQA-QL queries on data require to call the `value` function for each projection. Moreover, SOQA-QL doesn't provide all the useful operators of the object-oriented paradigm introduced in SQL99 like path expressions or collection manipulation. Last, SOQA-QL is a platform independent language whereas *OntoQL* is a language for OBDBs. As a consequence, *OntoQL* assumes that the data queried are stored in an OBDB and therefore it addresses some database problems such as query optimization or data definition and manipulation specific to OBDBs that cannot be considered by SOQA-QL due to its platform independency.

8.2 Semantic Web exploitation languages

Over the last years, many semantic web query languages have been proposed. Recently, a survey [20] classifies these languages into six categories with three main categories:

1. the SPARQL [21] category which groups query languages considering all data, both ontologies and their instances, as a set of triples;
2. the RQL [6] category which gathers query languages that make the distinction between the ontology and the data information (ontology instances). These languages provide operators to exploit the subsumption hierarchies of classes/properties and to combine data and schema querying;
3. the deductive languages (e.g, OWL-QL [7]) category for query languages expressing rules that define how new data can be derived from existing ones and thus be in the answer of a query.

OntoQL shares many characteristics with the second category. Indeed, like these languages, *OntoQL* offers the possibility to query ontologies, instances and both ontologies and instances but it does not offer rule based reasoning. However, contrary to these languages, *OntoQL* presents the following characteristics:

- *SQL Upward Compatibility. OntoQL* extends the SQL syntax and semantics. Thus, it has the benefits of SQL and it can be implemented as additional components of existing ORDBMS.
- *Schema manipulation.* In a lot of semantic integration approaches, the manipulation of the structure of the data is useful. *OntoQL* allows retrieving, creating, altering and dropping the schema of the data thanks to the possibility left to manage the metadata in both of the instance part or of the ontology part. Moreover, *OntoQL* uses this schema for query optimization;
- *Exploitation of multi-lingual definitions.* Concepts describe by an ontology may be associated with a linguistic representation in different natural languages. Using *OntoQL*, one can retrieve this representation and express queries in different natural languages.
- *Ontology model independency. OntoQL* is based on a core ontology model which can be extended to take into account specific features of a given ontology model. The RDF meta-model may also be extended. However, query

languages such as RQL restrict this extension to specializing the meta-classes `rdfs:Class` (the class of all classes) and `rdfs:Property` (the class of all of properties) to ensure a clear separation of the three abstraction layers of RDF and RDFS (data, ontologies and meta-schema). There is no such restriction with *OntoQL*. As a consequence, new attributes (e.g, comment, remark, illustration) and new entities (e.g, document, restriction) may be added and managed using *OntoQL*.

Regarding the expressive power, *OntoQL* doesn't allow to express query without specifying the search scope (the `FROM` clause is mandatory) and doesn't support yet the multi-instanciation capability. However, *OntoQL* is equipped with *grouping operators* (`GROUP BY`), *sorting operator* (`ORDER BY`) and *collection manipulation* operators not yet provided by semantic web query languages [22].

9 Conclusion

In this paper, we have formally presented an OBDB data model called OntoDB. This model differs from classical database models as well as other OBDB data models propositions. The need for a new exploitation language to manage this OBDB data model was a result of this constatation.

As a consequence, we proposed a formal algebra of operators together with the definition of the *OntoQL* database exploitation language for managing OBDBs. We have shown on some query examples how this language exploits the characteristics of the OBDB data model to support the multilingual querying of OBDBs at the ontology, content and both ontology and content levels. As a further step, we have defined an operational approach implementing these operators on top of a relational database model. The interested reader is invited to see the demonstrations of this prototype available at
http://www.plib.ensma.fr/plib/demos/ontodb/index.html.

OntoQL differs from the semantic web languages in the sense that it originates from databases approaches. It is built on top of RDBs and ORDBs preserving an upward compatibility and getting benefits of the power of database approaches keeping the possibility to exploit Web Semantic data.

For the future we plan to work in two directions related to database and to the semantic web. From a database perspective, it is important to study the query optimization on large databases and then the scalability of the OntoQL implementations in order to address large sets of data. Optimizations on the algebra operators and their composition shall be studied as well.

From a semantic web point of view, it it is planned to relax some assumptions made in the OBDB data model in order to offer an efficient storage capability for the instances described in the logic based approaches for ontologies like in OWL. The objective is to unify the proposition issued from the semantic web community which extensively use triples and descriptive logic and their derivatives, and the object orientation and database communities which use strong typing approaches.

References

1. Brickley, D., Guha, R.: RDF Vocabulary Description Language 1.0: RDF Schema. World Wide Web Consortium. (2004)
2. Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: OWL Web Ontology Language Reference. World Wide Web Consortium. (2004)
3. Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., Tolle, K.: The ics-forth rdfsuite: Managing voluminous rdf description bases. In: SemWeb. (2001)
4. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying rdf and rdf schema. In: International Semantic Web Conference. (2002) 54–68
5. Pan, Z., Heflin, J.: Dldb: Extending relational databases to support semantic web queries. In: PSSS. (2003)
6. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: Rql: a declarative query language for rdf. In: WWW. (2002) 592–603
7. Fikes, R., Hayes, P.J., Horrocks, I.: Owl-ql - a language for deductive query answering on the semantic web. J. Web Sem. **2** (2004) 19–29
8. Bellatreche, L., Pierra, G., Xuan, D.N., Dehainsala, H., Aït-Ameur, Y.: An a priori approach for automatic integration of heterogeneous and autonomous databases. In: DEXA. (2004) 475–485
9. Pierra, G.: Context-explication in conceptual ontologies: Plib ontologies and their use for industrial data. Journal of Advanced Manufacturing Systems (2004)
10. Jean, S., Pierra, G., Aït-Ameur, Y.: Domain ontologies: a database-oriented analysis. In: Web Information Systems and Technologies (WEBIST'2006). (2006) 341–351
11. Harris, S., Gibbins, N.: 3store: Efficient bulk rdf storage. In: PSSS. (2003)
12. Agrawal, R., Somani, A., Xu, Y.: Storage and querying of e-commerce data. In: VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc. (2001) 149–158
13. Jean, S., Pierra, G., Aït-Ameur, Y.: Ontoql: an exploitation language for obdbs. In: VLDB PhD Workshop. (2005) 41–45
14. Shaw, G.M., Zdonik, S.B.: A query algebra for object-oriented databases. In: ICDE. (1990) 154–162
15. Jean, S., Aït-Ameur, Y., Pierra, G.: Querying ontology based databases. the ontoql proposal. In: 18^{th} International Conference on Software Egineering and Knowledge Engineering (SEKE'2006). (2006) 166–171
16. Douglas, K., Douglas, S.: PostgreSQL. New Riders Publishing (2003)
17. Lakshmanan, L.V.S., Sadri, F., Subramanian, I.N.: Schemasql - a language for interoperability in relational multi-database systems. In: VLDB. (1996) 239–250
18. Litwin, W., Abdellatif, A., Zeroual, A., Nicolas, B., Vigier, P.: Msql: A multidatabase language. Inf. Sci. **49** (1989) 59–101
19. Ziegler, P., Sturm, C., Dittrich, K.R.: Unified querying of ontology languages with the sirup ontology query api. In: BTW. (2005) 325–344
20. Bailey, J., Bry, F., Furche, T., Schaffert, S.: Web and semantic web query languages: A survey. In: Reasoning Web. (2005) 35–133
21. W3C: Sparql. visited on (2005) retrieved from http://www.w3.org/TR/rdf-sparql-query/.
22. Haase, P., Broekstra, J., Eberhart, A., Volz, R.: A comparison of rdf query languages. In: SemWeb. (2004)