# OntoQL: an Alternative to Semantic Web Query Languages

Stéphane Jean

*LIAS-ENSMA and University of Poitiers*
*BP 40109, 86961 Futuroscope Cedex, France*
*jean@ensma.fr*

Yamine Aït-Ameur

*IRIT/INP-ENSEEIHT*
*Toulouse, FRANCE*
*yamine@enseeiht.fr*

Guy Pierra

*LIAS-ENSMA and University of Poitiers*
*BP 40109, 86961 Futuroscope Cedex, France*
*pierra@ensma.fr*

Ontologies are used in several application domains for representing knowledge. The defined approaches differ according to the type of addressed ontology (conceptual or linguistic) and to the used ontology model (e.g., OWL or PLIB). Several languages have been proposed to manipulate ontologies and their instances, especially in the Semantic Web domain. However these languages are often specific to a given ontology model, they focus on conceptual ontologies and they are not compatible with database exploitation languages. We address these three problems in this paper by proposing the OntoQL language. This language has three main original characteristics: (1) OntoQL is based on a core ontology model composed of the shared constructors of ontology models. This core ontology model can be extended by the language itself, (2) OntoQL queries can be expressed with different natural languages features using the linguistic layer of an ontology, and (3) OntoQL is fully compatible with SQL enabling a smooth integration between SQL queries of classical database applications and ontological queries. As a theoretical validation of this language, we present the algebra of operators that sets up its formal semantics. On the operational side, we describe the implementation of OntoQL on the OntoDB database and we illustrate the interest of this language by reporting several applications where this language has been extensively used and proved powerful.

## 1. Introduction

The intensive use of computer applications led to the availability of a huge amount of data in different formats (text, videos, signals, etc.) exchanged over networks like the web or specific enterprise networks. Classical access services based on representation formats or application and system-dependent access services have reached their limits. Indeed, the former is usually used for exchange purpose while the latter is application or system-dependent.

Providing high quality access services to these data is a big challenge. By quality, one may think of (i) reducing access time and space occupancy, (ii) displaying these data on any device (a smartphone for example), (iii) providing a user and/or context sensitive access services, (iv) providing a semantic-based access interface or, (v) other characteristics such as security, availability, accuracy, freshness, etc. All the previously cited quality characteristics have drawn the attention of several research work in the past recent years.

In this context, we have identified three relevant requirements we claim to handle by the proposal contained in this paper. *A first requirement* deals with the quality criteria related to the availability of a semantic-based access interface. Indeed, one major concern consists in supplying a set of access services that use high level domain knowledge close to the user knowledge, abstracting logical representations and exchange formats. In this setting, *ontologies* are good candidates that contribute to the design of such services.

Defined by T. Gruber [1] as an explicit specification of a conceptualization, an ontology is a model for representing the semantics of data. The explicit characteristics is fundamental; it makes it possible to manipulate semantic concepts of a domain of interest as first order objects. Nowadays, ontologies are used in a lot of diverse research and application domains including Natural Language Processing [2], Information Retrieval [3], Semantic Web [4], Databases [5], System Engineering [6], etc. According to the application domain, ontologies are defined in different ways. These different modeling approaches led to the availability of:

- *different types of ontologies.* Indeed, two main categories of ontologies have been identified in the literature: 1) *conceptual ontologies* aiming at defining the categories and properties of objects that exist in a given domain. These ontologies are useful in application domains like engineering where the domain of interest pre-exists, and 2) *linguistic ontology* aiming at representing the meaning of the words used in a particular universe of discourse. These ontologies are useful in application domains like Information Retrieval to explicit the meaning of words;
- *different ontology models.* Ontology models are formal knowledge models supporting the definition of ontology concepts and relationships as well as the corresponding reasoning mechanisms. Here again, ontology models have been designed according to their application domain. Note that some of these models become a reference and are standardized. OWL [7] ontology model for the Semantic Web and PLIB [8] ontology model for engineering are key examples of such ontology models.

In several domains where ontologies have been set up, the amount of knowledge and data exploited and manipulated becomes important and thus, persistence mechanisms are needed. Ontological data resulting from the use of ontologies need such mechanisms as well. *Offering a persistent framework for ontological data constitutes the second identified requirement.* To deal with persistence, Ontology-Based

Databases (OBDBs) [9] have been introduced. OBDBs are particular database systems that store both ontologies and the associated data. The development of OBDBs followed the same path as the one of ontology models. Indeed, two types of OBDBs have been defined according to the supported ontology model. Triplestores (e.g, Jena [10] or Oracle Spatial and Graph [11]) have been defined for OWL-based ontologies while systems such as OntoDB [9] or the Library Management Systems developed by Toshiba Corp [12] have been developed for PLIB-based ontologies.

Finally, all the previously addressed application domains use ontologies to provide users with services capable to exploit data at a higher conceptual level provided by the ontologies. This exploitation is performed thanks to the establishment of a link between the concepts of an ontology and the exploited data. This link may take several forms like annotation [13], subsumption [14], classification [15], indexation [16], etc. When setting up an OBDB to handle ontological data, indexation takes an important place in the design and exploitation of these data. Indeed, the use of semantic indexation i.e., the use of ontology concepts to index the ontological data, gave rise to semantic services that allow users to retrieve data from their semantic characterisation. These services are defined thanks to the availability of ontology exploitation languages. During the last decade a number of ontology query languages have been proposed especially in the Semantic Web context (see [17, 18] for a survey) and the SPARQL query language [19] has been accepted as the standard Semantic Web Query language. If these languages support queries on both ontologies and their instances, they are often specific to a given ontology model, they focus on conceptual ontologies and they do not keep compatibility with the usual Database Management System (DBMS) languages. *So, as a third requirement, we identify the need of a complete ontology exploitation language.*

Therefore the aim of our work is to design an exploitation language for ontologies and their instances, namely ontological data, that fulfills the previous identified requirements. In other words, the language shall 1) support different ontology models, 2) exploit characteristics of the different types of ontologies and 3) keep compatibility with the standard DBMS language (SQL). The OntoQL language, we initially proposed in [20, 21, 22], fulfills the previous requirements. During the last years, this language has been successfully used in many applications experiences both in research and industry. Compared to our previous work, this paper gives a complete and up-to-date definition of this language and describes some applications and experiences with this language.

This paper is structured as follows. Next section describes our classification of ontologies that has impacted the definition of OntoQL. We identify different types of ontologies and their combination in a layered model. We also show that ontology models share common constructors and conclude this analysis by clarifying a set of requirements for an exploitation language for ontologies and their instances. Section 3 uses these requirements to discuss advantages and shortcomings of existing languages. The need of a new language highlighted in this study leads us to the definition of the OntoQL language. Section 4 and Section 5 present the formal data model and an algebra designed for OntoQL. Furthermore, this language is presented in Section 6 through a set of examples. As a proof of concepts, Section 7 overviews the operational developments of the OntoQL engine. Finally, Section 8 summarizes the main results and introduces future work.

## 2. Analysis of the Ontology Notion

### 2.1. *Ontology Notion and Taxonomy*

From our point of view, an ontology is a *formal and consensual dictionary of categories and properties of entities of a domain and the relationships that hold among them* [23]. This definition encompasses three main characteristics of an ontology (1) *formal* i.e., it is based on a formal theory used to check the ontology consistency and to reason over the ontology-defined concepts and instances, (2) *consensual* as an ontology is agreed and shared by a community and (3) it can be *referenced* as each concept of an ontology has a universal identifier. Using this identifier, an ontology concept and the semantics it represents can be referenced from any environment, independently of the particular ontology where this concept was defined.

A criterion for classifying ontologies is their area of interest, if it consists of beings i.e., what does exist in the world, or of words i.e., how beings are apprehended and expressed in a particular natural language. This distinction leads to two categories of ontologies: *conceptual ontologies (CO)* and *linguistic ontologies (LO)* [24, 8]. A CO may only include *primitive concepts*, i.e. those concepts "for which we are not able to give a complete axiomatic definition" [1]. These ontologies, we called *Canonical Conceptual Ontologies (CCO)*, define a canonical vocabulary in which each information in the target domain is captured in a unique way without defining any synonymous constructs. A CO may also include *defined concepts*, i.e. those concepts for which the ontology provides a complete axiomatic definition by means of necessary and sufficient conditions expressed in terms of other concepts [1]. These ontologies we called *Non Canonical Conceptual Ontologies (NCCO)*, introduce new reasoning capabilities and they are useful to define mappings between different ontologies.

The three categories of ontologies introduced previously suggest a layered view of ontologies, we called the *onion model* of domain ontologies [23]. In this view, a kernel CCO provides a formal foundation to model and to exchange efficiently the knowledge of a domain. A NCCO layer extends the canonical vocabulary with concepts equivalence to encompass all concepts broadly used in the domain, thus extending inference capabilities. Finally, a LO layer adds the natural language representation of the CCO and NCCO concepts for person-system and person-person communication. The onion model shows that the capabilities of the different categories of ontologies can be combined. However, as we will see in Section 3, most of the existing ontology query languages do not exploit the three layers of the onion model. Another motivation of our work was to define a new ontology query language that was not specific to a particular ontology model. Indeed, as shown in next section, all ontology models share common constructors that can constitute the foundations of a generic ontology query language.

## 2.2. *Ontology Models*

In our work, we were primarily interested in the PLIB ontology model [8] defined for the engineering domain and in the RDF Schema (RDFS) [25] and OWL [7] ontology models defined for the Semantic Web. From the study of these ontology models, we have identified a set of shared constructors required to define CCOs. These ontologies are associated to a namespace in which concepts are defined as classes and properties. Classes are organized in a hierarchy using subsumption relationships. They are associated to properties which range may be a class or a datatype. Classes and properties can be referenced using an identifier independent of the underlying system (e.g., URI). They are described by names and definitions that may be given in different natural languages. Classes may have instances. Instances are characterized by their belonging classes and by the values of their properties.

Figure 1 presents an example, used throughout this paper, of a toy ontology defined with these shared constructors. It is inspired by the SIOC ontology [26]. This ontology is represented as a graph on the top of Figure 1. Its main concepts are the following. A forum (`Forum`) is hosted on a site (`Site`). It is managed by a moderator (`has_moderator`). Registered users (`User`) may subscribe to forums (`subscriber_of`) and write messages (`Post`) on these forums (`has_container`). A message may have several responses (`has_reply`). Instances of this ontology are represented in the bottom of Figure 1. URIs of instances are represented in an oval while literal values are represented in a rectangle. An instance of the `Post` class is described. The URI of this message ends with `post-sioc`. Its title and content are defined by literal values while its creator, its host forum and its responses are defined by referencing other instances.

If the studied ontology models share a set of constructors, they may differ in their underlying assumptions.

- *Open World Assumption (OWA) vs Close World Assumption (CWA).* PLIB makes the *CWA*: any statement that is not known to be true is false. This assumption is useful in the engineering context where a number of reference ontologies are already standardized and the knowledge can be considered as complete [8]. On the contrary, Semantic Web ontology models like RDFS or OWL are based on the *OWA*: any statement that is not known can be true. This assumption is more adapted to an open context like the Web.
- *Unique Name Assumption (UNA).* Under the *UNA*, if two objects have different identifiers, they are different. PLIB makes this assumption while RDFS and OWL do not. Without the UNA assumption, if two instances (or classes, or properties) have different identifiers, we may still need to derive by inference that they are the same.

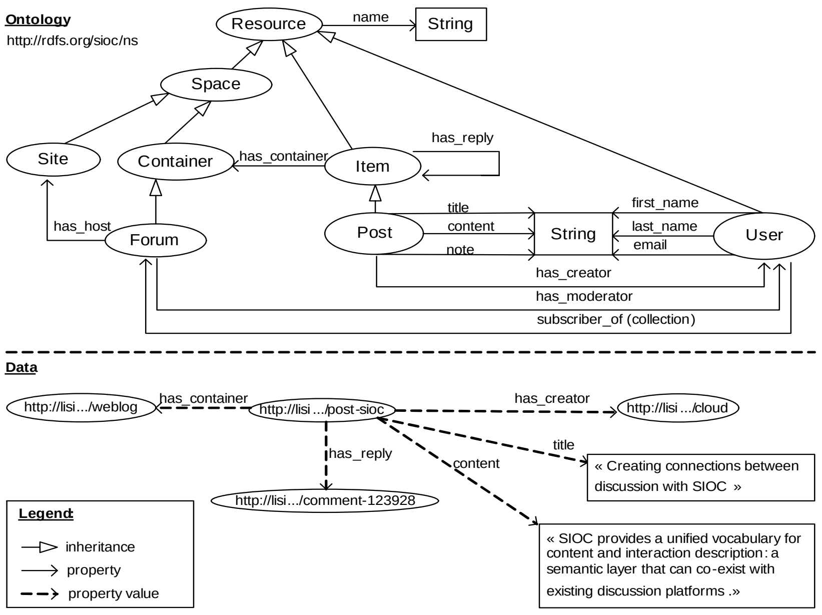


Fig. 1. A graphical representation of a toy ontology

- *Typing Assumption.* RDFS and OWL adopt a `Weak Typing Assumption`: an instance may belong to any number of non connected ontology classes. Contrariwise, PLIB adopts a `Strong Typing Assumption`: (1) each instance belongs to a single characterization class only and to all its subclasses, (2) each property is defined in the context of a class that defines its domain and (3) only the properties defined in the context of a class can be used for describing its instances.

Thus ontology models are complementary to define ontologies. They share common constructors to define CCOs and have specific constructors and assumptions according to the application domain. Starting from this analysis of ontology models and of the notion of ontology, several requirements for a language aiming at managing ontologies and their instances can be set up.

## 2.3. *Requirements for an Ontology Exploitation Language*

Ontology models share common constructors with a specific semantics (e.g., subsumption relationship). The language shall support these main constructors.

**Requirement 1 (*Management at the Semantic Level*)**

*The language shall offer operators or reasoning mechanisms supporting the exploitation of the usual semantics of ontology models.*

As the onion model shows, ontologies have different layers (CCO, NCCO and LO) with different capabilities. The language shall exploit these different layers.

**Requirement 2 (*Management of the Different Layers of an Ontology*)**

*The language shall support the definition and exploitation of non canonical concepts as well as linguistic definitions of concepts possibly defined in different natural languages.*

Ontology models are complementary to define ontologies. In addition to core constructors available in all ontology models, these models provide specific constructors that may be useful according to the application domain. The language shall not be defined for a specific ontology model.

**Requirement 3 (*Generic Ontology Exploitation Language*)**

*The language shall support core constructors of ontology models and provide mechanisms to support specific constructors.*

Usual exploitation languages such as SQL are sub-divided into several languages to define, manipulate and query data. These different sub-languages shall be available to manipulate ontologies and their instances.

**Requirement 4 (*Full Management of Ontologies and Instances*)**

*The language shall provide Ontology and Instances Definition, Manipulation and Query Languages.*

If ontology and instances are managed in a relational database, one may want to combine SQL queries with ontological queries. As a consequence, a seamless integration with SQL is required.

**Requirement 5 (*SQL Compatibility*)**

*The language shall preserve SQL compatibility.*

We use these requirements as criteria to evaluate existing languages.

## 3. Related Work: the Need of a New Ontology Exploitation Language

### 3.1. *Existing Ontology Exploitation Languages*

Several languages have been proposed to manage ontologies and their instances especially in the Semantic Web context [17]. In this section we review the most relevant languages and give a positioning according to the previously defined requirements.

#### 3.1.1. *SPARQL*

SPARQL [19] is a W3C recommandation widely used in the Semantic Web community. An update language is also available [27]. SPARQL is a graph-matching query language. A query consists of a pattern (a set of triples with variables and filters) defined in a `WHERE` clause. This pattern is matched with a data source, and the values obtained from this matching are processed in the `SELECT` clause to compute the answer.

*Example.* Retrieve instances of the `Item` class[a] of the SIOC ontology.

```
SELECT ?i WHERE {?i rdf:type sioc:Item}
```

*Semantics.* The triple in the `WHERE` clause introduces the variable `?i` (a variable is prefixed by `?`) to iterate over instances of the `Item` class. This variable is specified in the `SELECT` clause to return its values.

As the previous example shows, SPARQL can be used to retrieve instances of a class. However, the result of the previous query depends on the implementation. Indeed, SPARQL is defined for matching RDF graphs with *simple entailment* i.e., only by using the RDF triples explicitly represented. Thus, if for a class `C`, a triple `(i, rdf:type, C)` is represented for each direct or indirect instance of `C`, then the previous query returns also all the instances of the class `Post`, subclass of `Item`. On the contrary, if only a triple `(i, rdf:type, C)` is represented for each direct instance of `C`, then the query returns only direct instances of `C`.

SPARQL can also be extended to other forms of entailment such as RDFS-entailment i.e., the full set of triples that a RDFS description entails. In this case, the previous query will always return all direct and indirect instances of `Item`. However, if only direct instances must be returned, SPARQL do not provide an explicit operator to retrieve them. As a conclusion, ontological queries can be expressed but their returned results depend on the represented triples or on their interpretation.

Finally, even if a SPARQL query has `SELECT` and `WHERE` clauses similar to a SQL query, this language is adapted to RDF querying and it does not provide a smooth integration with SQL queries used in database applications. To query relational data with SPARQL, one needs first to publish relational data as RDF triples data with a tool such as D2RQ [28].

### 3.1.2. *RQL*

RQL [29] is based on a functional approach similar to the OQL object-oriented language [30]. Simple RQL queries consist of a function call. These functions exploit the usual semantics of ontology models. For example, the query `SubClassOf(Post)` retrieves all subclasses of the `Post` class. More elaborate RQL queries can be defined using a traditional `SELECT-FROM-WHERE` syntax. The `FROM` clause introduces path expressions (with variables) built from a set of predefined basic path expressions and operators (e.g., `C{X}` is a basic path expression that introduces a variable `X` on

[a]For readability and conciseness, we use names instead of URIs and omit namespaces definition.

all instances of the class `C`). The `WHERE` clause is used to define conditions on variables introduced in the `FROM` clause. Finally, the `SELECT` clause defines the selected variables in the result.

*Example.* Retrieve all instances of the `Item` class of the SIOC ontology.

```
SELECT I FROM sioc:Item{I}
```

*Semantics.* The `FROM` clause introduces the variable `I` on all (direct and indirect) instances of the `Item` class. The `SELECT` clause projects the URIs of these instances. To retrieve only direct instances, the `Item` class must be prefixed with ˆ (i.e., `^Item`)

RQL has an update language named RUL [31] and a view language named RVL [32] which can be used to represent non canonical concepts such as OWL restrictions. The data model of RQL is based on RDFS and thus it is composed of the `Class` and `Property` constructors. This ontology model can be extended by specialization of these two constructors. But, a constructor can not be added if it does not inherit from `Class` or `Property`. This constraint forbids the definition of a number of constructors such as the `Document` constructor of PLIB (to describe a concept by a document) or the `Ontology` constructor of OWL (to regroup all concepts defined in an ontology). Moreover, the RQL syntax is close to the one of object-oriented languages but it does not preserve SQL compatibility.

### 3.1.3. *SOQA-QL*

SOQA-QL has been designed in the context of the SIRUP project [33] aiming at automatically integrating heterogeneous data sources according to user needs. Its main feature is to provide access to an ontology and its instances represented in various ontology models. This characteristic is based on the definition of a core ontology model named *SOQA Ontology Meta Model.* It contains the main constructors of different ontology models as described in section 2.2: ontologies, classes, properties, relationships, methods and instances. SOQA-QL queries follow the usual `SELECT-FROM-WHERE` syntax to retrieve characteristics of ontology components. As the following example shows, a set of functions can be called in SOQA-QL for complex queries.

*Example.* Retrieve the name and documentation of the direct super-classes of `Post`.

```
SELECT name, documentation FROM DirectSuperConcepts(sioc:Post)
```

*Semantics.* The `DirectSuperConcepts` function applied to `Post` returns all the direct super-classes of this class. The `SELECT` clause returns the name and documentation (characteristics defined in the SOQA Ontology Meta Model) of these classes.

Thus, to query ontologies, SOQA-QL follows the syntax and semantics of SQL. However, it is necessary to use the `Instances` and `Value` functions for querying the ontology instances. As a consequence, SQL queries can not be directly integrated with ontological queries. Moreover, SOQA-QL does not support non canonical concepts and partially linguistic information as multilingual definitions defined in an ontology can not be used. Finally, the SOQA Ontology Meta Model is not extendable with new constructors.

As the above analysis shows, the aforementioned languages present some limitations to support a uniform and shared manipulation of ontologies and their instances. These limitations are highlighted in Table 1. They motivated us to propose a new ontology exploitation language, we name OntoQL.

| **Requirements** | **SPARQL** | **RQL** | **SOQA-QL** |
|---|---|---|---|
| Management at the Semantic Level | RDF Triples (entailment) | Yes | Yes |
| Management of the Different Ontology Layers | CCO/LO | CCO/NCCO | CCO |
| Generic Ontology Exploitation Language | RDF | RDFS | Fixed Core Model |
| Full Management of Ontologies and Instances | Query /Update | Query/Update /View | Query |
| SQL Compatibility | No | No | No |

Table 1. Fulfilled requirements by the main ontology query languages

## 3.2. *Other Related Work*

In addition to the previous analysis of existing query languages for ontologies, we position our approach with respect to several work that aim at combining DBMS and ontologies.

*Ontology extraction from a persistent corpus of documents in a DBMS.* Recently, Garcia-Alvarado and Ordonez have proposed the *ONTOCUBO* system [34] built on their previous work on ONTOCUBE [35] and CUBO [36]. This system allows users to automatically extract an ontology from a corpus of documents where each document is characterised by a set of keywords. This extraction process identifies two steps: (1) *concept extraction* by analysing the most frequent keywords, generating combinations of these keywords and keeping the most relevant combinations using statistical measures and (2) *ontology building* by identifying *is_a* and *has_a* relationships between the extracted concepts of the previous step. The resulting ontology can be modified by an expert, then it is summarized by building an OLAP cube. The dimensions of this cube are the ontology classes. It aggregates a set of desired measurements. For managing large corpus efficiently, ONTOCUBO has been completely implemented within a DBMS. ONTOCUBO together with the OntoQL language, proposed in this paper, are complementary. On the one hand, OntoQL assumes the existence of an ontology stored inside a DBMS. This ontology can be produced with ONTOCUBO and plays the role of a conceptual model. On the other hand, ONTOCUBO does not offer a declarative language to modify and query the resulting ontology. The OntoQL language has been designed for this purpose.

*Ontology extraction from a relational database.* Many approaches have been proposed to build an ontology from a relational database (see [37] for a survey). For example, Astrova [38] proposes a rule-based approach which specifies how constructors of the relational model are mapped to the OWL metamodel. These rules are then applied on a given relational database to produce the resulting ontology. Like ONTOCUBE, these approaches are complementary with OntoQL.

*Mapping a relational database to an ontology.* Several approaches such as D2RQ [28] propose to map a relational database to an existing ontology. Using this approach, the relational data can be directly accessed from an ontology query language such as SPARQL or OntoQL.

*Enhancing DBMS data with ontologies.* Wiegand [5] has given concrete examples and use cases of the potential of ontologies to enhance geographical data stored inside a DBMS. More precisely, examples are given for data organization, query expansion, feature-based modelling and a linked knowledge base with explicit spatial relations. These examples have been implemented within the Oracle DBMS using SQL table functions to query RDF data [11]. These examples could also be implemented in OntoQL and extended with the specific features of OntoQL (e.g., to perform query expansion in different natural languages).

*Ontology-based design of Semantic Data Warehouse (SDW).* Several approaches have proposed to design a SDW using an ontology. For example, Bellatreche et al. [14] have proposed a SDW design methodology covering the steps of its life cycle. In this design methodology, the required data for each design step need to be stored in the SDW. This approach has been successfully implemented on top of OntoDB using the OntoQL language proposed in this paper.

## 4. The OntoQL Data Model

Before presenting the data model of OntoQL, it is necessary to precise our assumptions. Indeed, as we have seen in section 2.2, ontology models share common constructors but differ on their assumptions. Since our work mainly targets the engineering domain, we have chosen to follow the PLIB assumptions. Thus the close-world and unique names assumptions are made. Under these assumptions, we have designed the data model of OntoQL. It is composed of two related parts: *ontology* and *content*. Instances are managed in the content part while ontologies are managed in the ontology part.

### 4.1. ***Ontology.***

The ontology part represents ontologies as instances of an ontology model. It is formally defined by a 7-tuple as $< \mathtt{E}, \mathtt{OC}, \mathtt{A}, \mathtt{SuperEntities}, \mathtt{TypeOf}, \mathtt{AttDomain}, \mathtt{AttRange}, \mathtt{Val} >$.

- $\mathtt{E}$ 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 ontology concepts (classes, properties ...). They have a unique identifier.
- `A` is the set of attributes describing each ontology concept.
- $\mathtt{SuperEntities} : \mathtt{E} \rightarrow 2^{\mathtt{E}}$[b] 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.
- $\mathtt{TypeOf} : \mathtt{OC} \rightarrow \mathtt{E}$ associates to each concept of an ontology the lower (strongest) entity in the hierarchy it belongs to.
- $\mathtt{AttributeDomain}, \mathtt{AttributeRange} : \mathtt{A} \rightarrow \mathtt{E}$ define respectively the domain and the range of each attribute.
- $\mathtt{Val} : \mathtt{OC} \times \mathtt{A} \rightarrow \mathtt{OC}$ gives the attribute value of an ontology concept.

The OntoQL 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 $\{\mathtt{o}_1, \ldots, \mathtt{o}_n\}$ is an object of this type (the $\mathtt{o}_i$'s are objects of type T). The $\mathtt{Tuple}[< (\mathtt{A}_1, \mathtt{T}_1), \ldots, (\mathtt{A}_n, \mathtt{T}_n) >]$ parameterized type creates relationships between objects. It is constructed by providing a set of attribute names ($\mathtt{A}_i$) and attribute types ($\mathtt{T}_i$). $\mathtt{Tuple}[< (\mathtt{A}_1, \mathtt{T}_1), \ldots, (\mathtt{A}_n, \mathtt{T}_n) >]$ denotes a tuple type constructed using the $\mathtt{A}_i$ attribute names and $\mathtt{T}_i$ attribute types. $< \mathtt{A}_1 : \mathtt{o}_1, \ldots, \mathtt{A}_n : \mathtt{o}_n >$ is an object of this type (the $\mathtt{o}_i$'s are objects of type $\mathtt{T}_i$). The `Tuple` type is equipped with the $\mathtt{Get\_A_i\_value}$ functions to retrieve the value of an attribute $\mathtt{A}_i$ in the `Tuple` object `o`. The application of this function is abbreviated using the dot-notation ($\mathtt{o.A}_i$).

`E` provides the predefined entities `C` and `P`. Instances of `C` and `P` are respectively the classes and properties of the ontologies and the types that can be built from them. Entity `C` defines the attribute $\mathtt{SuperClasses} : \mathtt{C} \rightarrow \mathtt{SET[C]}$ and entity `P` defines the attributes $\mathtt{PropDomain} : \mathtt{P} \rightarrow \mathtt{C}$ and $\mathtt{PropRange} : \mathtt{P} \rightarrow \mathtt{C}$. The description of these attributes is similar to the definitions given for `SuperEntities`, `AttributeDomain` and `AttributeRange` replacing entities by classes and attributes by properties. A global super-class `Root` is predefined to initialize the hierarchy.

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 consti-

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

tute the kernel of the ontology models we have considered. Notice that this kernel is defined with the various constructors shared by the common standard ontology models PLIB, RDFS and OWL. As discussed in section 6.2, user-defined entities (e.g., an illustration) and attributes (e.g., a remark) may be added to this kernel in order to take into account the specific features of an ontology model.

*Example.* The following statements illustrate the formal data model of the ontology part. It presents an extract of the ontology part of the chosen illustrative ontology of Figure 1.

- `E = { C, P }`.
- `A = { oid, code, name, definition, PropDomain, ... }`.
- `AttScope(PropDomain) = P` ; `AttRange(PropDomain) = C`.
- `C = { Post, User, ... }`.
- `P = { title, content, note, has_creator, has_reply, ... }`.
- `PropScope(URI) = Post` ; `PropRange(URI) = String`.

## 4.2. *Content.*

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

- `EXTENT` is a set of *extensional definitions* of ontology classes.
- `I` is the set of instances. Each instance has an identity.
- $\mathtt{TypeOf} : \mathtt{I} \to \mathtt{EXTENT}$ associates to each instance the extensional definition of the class it belongs to (collection of its instances).
- $\mathtt{SchemaProp} : \mathtt{EXTENT} \to 2^{\mathtt{P}}$ gives the properties used to describe the instances of an extent (the set of properties that have a value for its instances).
- $\mathtt{Val} : \mathtt{I} \times \mathtt{P} \to \mathtt{I}$ 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.

*Example.* Extract of the content part of our example ontology.

- `Extent = { Extent_Post }`.
- `I = { $1 }`, where `$1` is the instance whose `URI` ends with `post-sioc`.
- `TypeOf(I) = Extent_Post`.
- `SchemaProp(Extent_Post) = { title, content, has_creator, has_reply, has_container }`.
- `Val(I, has_creator) = $1`, where `$2` is the instance whose `URI` ends with `cloud`.

## 4.3. *Linking Ontology and Content Parts*

The relationship between an ontology and its instances (content) is defined by the partial function $\mathtt{Nomination} : \mathtt{C} \to \mathtt{EXTENT}$. It associates a definition by intension with a definition by extension of this class. In the previous example, `Nominiation(Post) = Extent_Post`. A class without extensional definition is 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: $(\mathtt{propDomain}^{-1}(\mathtt{c}) \supseteq \mathtt{SchemaProp}(\mathtt{nomination}(\mathtt{c})))$.

As a first step for the design of an exploitation language for ontologies and its instances, we build a query algebra for this data model. The interest of this algebra is the ability to express the full extraction operators.

## 5. The OntoQL Query Algebra: OntoAlgebra

Since the OntoQL data model uses extensively object-oriented database (OODB) features, we suggest to specialize, extend and reuse the operators issued from the *ENCORE* algebra [39].

### 5.1. *Main operators of OntoAlgebra*

The signatures of the operators defined on the OntoQL data model belong to $(\mathtt{E} \cup \mathtt{C}) \times 2^{\mathtt{OC} \cup \mathtt{I}} \rightarrow (\mathtt{E} \cup \mathtt{C}) \times 2^{\mathtt{OC} \cup \mathtt{I}}$. The main operators of this algebra are *OntoImage*, *OntoProject*, *OntoDupEliminate*, *OntoSelect*, *OntoOJoin*, *OntoNest* and ***. The complete definition of this algebra being outside the scope of this paper, solely these operators, restricted to the content part, are formally presented below. Their signature is $\mathtt{C} \times 2^{\mathtt{I}} \rightarrow \mathtt{C} \times 2^{\mathtt{I}}$. The interested reader can refer to [21, 40] for the complete definition of this algebra. These papers show that 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, several properties occurring in the function parameter may not be valued in the extensional definition of an ontology 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 used in 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). Thus we have chosen to interpret `UNKNOWN` as a NULL in SQL. With the introduction of `OntoVal`, the semantics of `OntoImage` is defined by:

$$\begin{aligned}&\mathtt{OntoImage}(\mathtt{T},\{\mathtt{i}_1,\dots,\mathtt{i}_n\},\mathtt{f}) = \\ &\qquad(\mathtt{PropRange}(\mathtt{f}),\{\mathtt{OntoVal}(\mathtt{i}_1,\mathtt{f}),\dots,\mathtt{OntoVal}(\mathtt{i}_n,\mathtt{f})\})\ .\end{aligned}$$

**- 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:

$$\begin{aligned}&\mathtt{Project}(\mathtt{T},\mathtt{I}_t,\{(\mathtt{A}_1,\mathtt{f}_1),\dots(\mathtt{A}_n,\mathtt{f}_n)\}) = \\ &\quad(\mathtt{Tuple}[<(\mathtt{A}_1,\mathtt{PropRange}(\mathtt{f}_1)),\dots,(\mathtt{A}_n,\mathtt{PropRange}(\mathtt{f}_n))>], \\ &\qquad\{<\mathtt{A}_1:\mathtt{OntoVal}(\mathtt{i},\mathtt{f}_1),\dots,\mathtt{A}_n:\mathtt{OntoVal}(\mathtt{i},\mathtt{f}_n)>|\mathtt{i}\in\mathtt{I}_t\})\ .\end{aligned}$$

It returns the type of elements together with the set of corresponding values.
**- OntoDupEliminate.** It is used with the `OntoImage` and `OntoProject` operators to eliminate duplicates in a query result. It is based on the equality of two elements of `I`. Two elements are equals if one of the following conditions is fulfilled: (1) they are both collections of the same cardinality and there is a one-to-one equality between their members , (2) they are two tuples of the same arity and their corresponding attribute values are equal, (3) they have the same value of the same atomic type, (4) they are two ontology instances with the same identifier. With these definitions, `OntoDupEliminate` is defined by:

$$\mathtt{OntoDupEliminate}(\mathtt{T},\mathtt{I}_t) = (\mathtt{T},\mathtt{I}_r)\ .$$

$\mathtt{I}_r$ is a collection without duplicates (`Set[T]`) built from the collection $\mathtt{I}_t$.
**- OntoSelect.** It creates a collection of objects satisfying a selection predicate. Its signature is $\mathtt{C}\times 2^{\mathtt{I}}\times\mathtt{Predicate}\rightarrow\mathtt{C}\times 2^{\mathtt{I}}$ and its semantics is defined by:

$$\mathtt{OntoSelect}(\mathtt{T},\mathtt{I}_t,\mathtt{pred}) = (\mathtt{T},\{\mathtt{i}|\mathtt{i}\in\mathtt{I}_t\wedge\mathtt{pred}(\mathtt{i})\})\ .$$

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`.
**- OntoOJoin.** It creates relationships between elements of two collections. This operator is similar to a Θ-join in the relational algebra i.e., the result of this operation consists of all combinations of elements of the two collections that satisfy the predicate Θ (denoted `pred` in the following). It is defined by:

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

In this definition, `r` and `t` must be valid input of `pred`.
**- OntoNest.** It is used to represent tuples as a nested relation. The comparisons

made by this operator are based on the equality relationship previously defined for the `OntoDupEliminate` operator. It is defined by:

$$\begin{aligned}&\mathtt{OntoNest}(\mathtt{TUPLE}[<(\mathtt{A_1},\mathtt{T_1}),\ldots,(\mathtt{A_i},\mathtt{T_i}),\ldots,(\mathtt{A_n},\mathtt{T_n})>],\mathtt{I_t},\mathtt{A_i}) = \\ &\qquad(\mathtt{TUPLE}[<(\mathtt{A_1},\mathtt{T_1}),\ldots,(\mathtt{A_i},\mathtt{SET}[\mathtt{T_i}]),\ldots,(\mathtt{A_n},\mathtt{T_n})>], \\ &\qquad\{<\mathtt{A_1}:\mathtt{s.A_1},\ldots,\mathtt{A_i}:\mathtt{t},\ldots,\mathtt{A_n}:\mathtt{s.A_n}>\mid \forall \mathtt{r}\in\mathtt{t}\ \exists \mathtt{s}\in\mathtt{I_t}.(\mathtt{s.A_i}=\mathtt{r})\})\end{aligned}$$

**- Operator *.** It is the explicit polymorphic operator to distinguish between queries on instances of a class $\mathtt{C}$ and instances of all the classes subsumed by $\mathtt{C}$ and denoted $\mathtt{C}^*$. It is based on the functions `ext` and `ext*`. $\mathtt{ext}:\mathtt{C}\to 2^{\mathtt{I}}$ returns direct instances of a class and $\mathtt{ext}^*:\mathtt{C}\to 2^{\mathtt{I}}$ its deep extent. If $\mathtt{c}$ is a class and $\mathtt{c_1},\ldots \mathtt{c_n}$ are the direct sub-classes of $\mathtt{c}$, `ext` and $\mathtt{ext}^*$ are derived recursively[c] by:

$$\mathtt{ext}(\mathtt{c}) = \mathtt{TypeOf}^{-1}(\mathtt{Nomination}(\mathtt{c}))\ .$$
$$\mathtt{ext}^*(\mathtt{c}) = \mathtt{ext}(\mathtt{c})\cup\mathtt{ext}^*(\mathtt{c_1})\cup\ldots\cup\mathtt{ext}^*(\mathtt{c_n})\ .$$

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

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

## 5.2. *Properties and Complexity of OntoAlgebra*

The *closure* and *relational completeness* properties are two important properties of query languages which have been used to characterize the SQL language but also RDF [41] and XML [42] query languages.

The closure property expresses composability; it requires that the result of an operator can be the input of another operator. OntoAlgebra is closed as all its operators return a collection of either class instances (which can be atomic values or tuples) or entity instances. This property can also be seen by the signatures of the operators which belong to $(\mathtt{E}\cup\mathtt{C})\times 2^{\mathtt{OC}\cup\mathtt{I}}\to(\mathtt{E}\cup\mathtt{C})\times 2^{\mathtt{OC}\cup\mathtt{I}}$.

The notion of *relational completeness* has been introduced by Codd [43] to characterize the expressive power of query languages. An algebra $A$ is relational complete if for all expressions built from the relational algebra, there is an equivalent expression built from the operators of $A$. OntoAlgebra has this property as its operators can be used to compute the different operations of the relational algebra:

- `OntoImage` and `OntoProject` are used to compute projections;
- `OntoSelect` is used to compute selections;
- `OntoOJoin` is used to compute cartesian product (with a predicate always true) and the different forms of joins;

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

- `OntoUnion` and `OntoDifference` are used to compute set operations.

Another important characteristic is the complexity of the operators. Similarly to the relational algebra [44], the complexity of the main operators of OntoAlgebra is defined in terms of the number of operations required and based on the input cardinality without considering physical implementation details.

- `OntoImage`, `OntoProject` and `OntoSelect` require to iterate on the instances of the input class (or entity). Thus, the complexity of these operators is *O(n)* where $n$ denotes the number of class instances.
- `OntoDupEliminate` and `OntoNest` require to iterate on the instances of a class for each instance of this same class. Thus, the complexity of these operators is $O(n^2)$. This complexity can be reduced to $O(n * log\ n)$ if the class instances are sorted on the identifier attribute and $O(n)$ if hashing techniques are used.
- `OntoOJoin` and set operations (`OntoUnion` and `OntoDifference`) require to iterate on the instances of a class for each instance of an other class. Thus the complexity of these operators is also $O(n^2)$ (which can be reduced as explained previously).

OntoAlgebra expressions and the `*` operator are a composition of OntoAlgebra operations and thus their complexities can be deduced from the complexity of the previous operators.

## 5.3. *Query optimization techniques based on OntoAlgebra*

Traditional optimization techniques defined for the relational algebra can be applied with OntoAlgebra. For example, the strategy consisting in pushing selections past joins is characterized by the following equivalence rule:

$$\mathtt{OntoSelect}(\mathtt{OntoOJoin}(\mathtt{T}, \mathtt{I_t}, \mathtt{R}, \mathtt{I_r}, \mathtt{pred}), \mathtt{pred_t}) \\ \Leftrightarrow \mathtt{OntoOJoin}(\mathtt{OntoSelect}(\mathtt{T}, \mathtt{I_t}, \mathtt{pred_t}), \mathtt{R}, \mathtt{I_r}, \mathtt{pred})$$

Besides traditional optimization techniques, optimizations based on partial evaluation techniques can be set up. Indeed, some of the properties defined on an ontology class may not have a value for the instances of this class. Thus it is not necessary to search the values of these properties. More formally, let `p` be a property, `C` a class, $\mathtt{I_c}$ its instances and `pred` a predicate in conjunctive normal form involving `p`, then:

$$\mathtt{p} \notin \mathtt{usedProperties}(\mathtt{nomination}(\mathtt{C})) \Rightarrow \mathtt{OntoSelect}(\mathtt{C}, \mathtt{I_c}, \mathtt{pred})) = \emptyset$$

This rule can also be defined for the `OntoOJoin` operator as it also involves a predicate.

*Example.* Figure 2 presents an example of an optimization based on the previous rule. This example assumes that the `User` class has a subclass named

`Administrator`. The property `email` is used (resp. not used) for describing the instances of the class `User` (resp. `Administrator`).

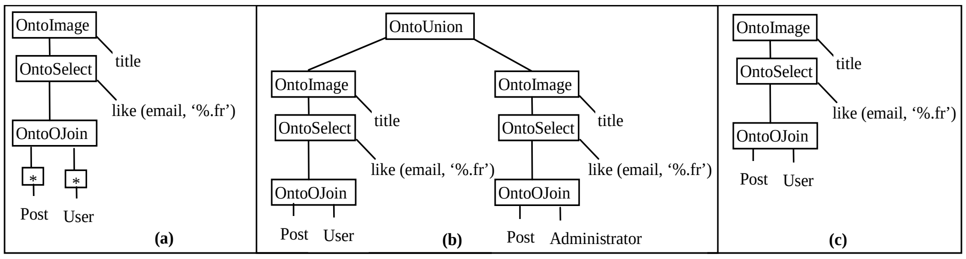


Fig. 2. Example of an optimization based on a partial evaluation technique

*Semantics.* Figure 2(a) presents the OntoAlgebra query tree of the initial query. This query searches for the title of the post written by users whose email ends with `.fr`. In Figure 2(b), the operator `*` is erased from the query plan by using a union operator as we have seen in its definition. The result is the union of two queries: one for the `User` class and one for the `Administrator` class. As the email, property is not used for describing the instances of `Administrator`, the second query can be erased from the query plan and thus we obtained the optimized query plan depicted in Figure 2(c).

Once OntoAlgebra is defined, it can be used to interpret OntoQL instructs. Next section shows the main constructs of the OntoQL language through its concrete syntax.

## 6. The OntoQL Language

The OntoQL language provides access to (1) ontology instances through its Data Definition, Manipulation, and Query Languages (Section 6.1), (2) ontologies through its Ontology Definition, Manipulation, and Query Languages (Section 6.2) and both ontologies and their instances (Section 6.3).

### 6.1. *The OntoQL Data Definition, Manipulation, and Query Languages (DDL, DML and DQL)*

**SQL compatibility**. OntoQL should preserve SQL compatibility in order to integrate OntoQL queries with SQL queries used in existing database application (requirement 5). A particularity of an ontology compared to a relational schema is that its components (classes and properties) have a universal identifier based on a namespace. Thus the distinction between the two data access levels (relational and ontological) is based on the availability of a namespace in identifiers. OntoQL supports two mechanisms for namespaces. The `USING NAMESPACE` clause of OntoQL specifies namespaces used in the scope of a query while the `SET NAMESPACE` OntoQL statement specifies namespaces for all statements executed in a session. If these clauses are not used, OntoQL processes queries as SQL statements.

**Access to the CCO layer**. The CCO layer is manipulated through three sublanguages of OntoQL: the DDL, DML and DQL. Since our algebra is based on an object-oriented algebra, we have chosen to define the OntoQL syntax starting from the SQL relational-object syntax [45] for manipulating user types. We have extended and adapted it to ontologies specific features. The following example illustrates these sublanguages.

*Example.* Create the class `Post` (statement 1) with its extent (statement 2) assuming that a post is only described by its title and creator. Insert an instance of this class (statement 3). Finally, find the last names of posts' creators (statement 4).

| | |
|---|---|
| `CREATE #Class Post UNDER Item (`<br>`DESCRIPTOR (#version = '001')`<br>`#Property (title String, content String, note Int,`<br>`has_creator REF(User) DESCRIPTOR (#version = 'v1')))` | Statement 1 |
| `CREATE EXTENT OF Post (title, has_creator)` | Statement 2 |
| `INSERT INTO Post (title, has_creator)`<br>`VALUES ('Title', 10)` | Statement 3 |
| `SELECT p.has_creator.last_name FROM ONLY(Post) AS p` | Statement 4 |

*Semantics.* Statement 1 creates the `Post` class with its properties and its description (`DESCRIPTOR` clause). After the `CREATE` keyword the type of class is specified by an entity. In the OntoQL syntax, entities and attributes are prefixed by `#` (e.g., `#version`) to distinguish them from ontology classes and properties (e.g., `title`). `#Class` and `#Property` are two built-in entities that represent respectively the set of classes and properties. Statement 2 creates the extent of the `Post` class. An extent of a class corresponds to a typed table in the SQL object-relational data model. However, contrary to a database schema which prescribes the attributes characterizing the instances of a user-defined type, an ontology only describes the properties that may be used to characterize the instances of a class. As a consequence the extent of class is only composed of the subset of properties that are really used to describe its instances. Statement 3 inserts an instance of the `Post` class with a statement similar to the one of SQL. Statement 4 is a DQL query similar to an SQL query. This langage includes object-oriented operators such as path expressions (e.g., `p.has_creator.lastname`) or type operators (e.g., `Only(Post)` is used to retrieve direct instances of the `Post` class).

**Access to the NCCO layer**. Ontology models include different types of NCCO constructs. Indeed, OWL supports NCCO class definitions using boolean expressions and restrictions and NCCO property definitions using inverse property. PLIB supports NCCO property definitions using derivation functions. Defining dedicated built-in operators for these operators raises many technically challenging problems, both theoretical and practical [46]. As a first step to manage NCCO constructs we have added a View Definition Language (VDL) to OntoQL. Using this language, a class is defined as non canonical using the `AS VIEW` keywords and its extent is computed using an OntoQL query.

*Example.* Create the `PostDupont` class defined as all messages of `Dupont`.

```
CREATE #Class PostDupont AS VIEW UNDER Post;
CREATE VIEW OF PostDupont AS
  SELECT * FROM Post AS p WHERE p.has_creator.last_name = 'Dupont';
```

*Semantics.* The first statement creates the `PostDupont` class. The position of this class in the hierarchy must be specified by the user. In our example, the `PostDupont` class is a non canonical class (`AS VIEW`) subsumed by the `Post` class (`UNDER Post`). The second statement creates the extent of this class using an OntoQL query. This query returns instances of `Post` having `Dupont` as author. These instances are those of the `PostDupont` non canonical class.

Non canonical classes are queried as canonical classes using the usual query rewriting mechanism. In contrast, adding, updating or deleting instances through a non canonical class brings back to the view update problem. Thus this operation runs only if the view corresponding to the extent of the class can be updated.
**Access to the LO layer**. In the previous examples, we have used identifiers of ontology classes and properties to manipulate them. However, in many real ontologies, these identifiers do not correspond to a name. For example, in the PLIB IEC ontology describing electronic components [47], the identifier of the class of resistances is `AAA089`. As a consequence, it is not straightforward to use identifiers in queries. OntoQL uses the LO layer of an ontology to overcome this difficulty.

*Example.* Retrieve the first and last names of users with identifiers and names in English and in French of the corresponding ontology classes and properties.

```
SELECT first_name, <=> SELECT "first name", <=> SELECT prénom,
       last_name              "last name"              nom
  FROM User              FROM User               FROM Utilisateur
                        USING LANGUAGE EN       USING LANGUAGE FR
     (A)                      (B)                      (C)
```

*Semantics.* Query `(A)` does not use the LO layer of an ontology. Classes and properties are referenced through their identifiers. Query `(B)` is equivalent to query `(A)` but it is written using names of the used classes and properties expressed in English. Quotes are used for names that include a space (e.g., `"first name"`). Query `(C)` is also equivalent to the previous queries but it is written with the names expressed in French.

In this section, we have shown that ontology instances are manipulated in OntoQL both at the logical level, keeping SQL compatibility, and at the ontological level according to the three layers of an ontology. Next section presents the capabilities of OntoQL to manipulate ontologies themselves.

### 6.2. *The OntoQL Ontology Definition, Manipulation, and Query Languages (ODL, OML and OQL)*

OntoQL shall support the manipulation of ontologies defined with constructors of different ontology models. To fulfill this requirement, OntoQL is based on a core ontology model that can be extended by the language itself. This core ontology model is composed of the shared constructors of different ontology models that we have identified in section 2.2. The following example illustrates the extension of this core ontology model.

*Example.* Add the `AllValuesFrom` OWL constructor to the core ontology model (statement 1). Then, create the class named `InvalidPost` of our example ontology with an instance (statement 2). Finally, search the `AllValuesFrom` restrictions defined on the `hasModifiers` property (statement 3).

| | |
|---|---|
| `CREATE ENTITY #OWLAllValuesFrom UNDER #Class (`<br>`#onProperty REF(#Property),`<br>`#allValuesFrom REF(#Class))` | Statement 1 |
| `INSERT INTO #OWLRestrictionAllValuesFrom`<br>`(#name[en], #name[fr], #onProperty, #allValuesFrom)`<br>`VALUES ('InvalidPost', 'Post invalide',`<br>`'hasModifiers', 'Post')` | Statement 2 |
| `SELECT #name[en], #allValuesFrom.#name[en]`<br>`FROM #OWLRestrictionAllValuesFrom`<br>`WHERE #onProperty.#name[en] = 'hasModifiers'` | Statement 3 |

*Semantics.* Statement 1 adds the `OWLAllValuesFrom` entity to our core ontology model as a sub-entity of the `Class` entity. This entity is created with two attributes, `onProperty` and `allValuesFrom`, which respectively take as values identifiers of properties and identifiers of classes. Statement 2 creates the `OWLAllValuesFrom` `InvalidPost` restriction with an `INSERT` statement. As we have seen previously, the definition of this class could also be made with a `CREATE` statement of the DDL. Indeed, syntactic equivalences are defined between OML and DDL statements. These two syntactic constructions are valid but in general the second one is more compact. Statement 3 is a query. As we can see OQL queries are similar to the one of the DQL except that entities and properties are used instead of classes and properties.

This section showed that OntoQL language can be used to query both ontologies instances or ontologies themselves. Next section shows how these capabilities are combined.

### 6.3. *Querying both Ontologies and Instances*

**From ontology to instances.** The OQL language part of OntoQL can be used to retrieve ontology classes and properties. In distributed applications where the network traffic should be minimized, retrieving simultaneously instances and values of these classes and properties instead of executing two queries may prove useful. To fulfil this need, iterators on the instances of a class identified at run-time are set up in OntoQL.

*Example.* Retrieve instances of the classes whose name in English ends with `Post`.

```
SELECT i.oid FROM #class AS C, C AS i WHERE C.#name[en] like '%Post'
```

*Semantics.* In this query, the `Post` and `InvalidPost` classes fulfil the condition of the selection. As a consequence this query returns instances identifiers of these two classes. And since `InvalidPost` is a subclass of `Post`, instances identifiers of the `InvalidPost` class are returned twice.

**From instances to ontology.** An ontology class hierarchy can be deep. Thus, when a DQL queries is executed to retrieve all instances of a given class, it is useful to retrieve the ontological description of the belonging class of each instance. To fulfil this need, OntoQL proposes the `typeOf` operator to retrieve the *basic class* of an instance i.e., the minorant class for the subsumption relationship of the classes it belongs to.

*Example.* Retrieve the English name of the basic class of `User` instances.

```
SELECT typeOf(u).#name[en] FROM User AS u
```

*Semantics.* This query iterates on the instances of the `User` class and on those of the `Administrator` class as well. For each instance, the query returns `User` or `Administrator` according to its member class.

In this section, we have seen the different sub-languages of OntoQL for defining, manipulating and querying ontologies and their instances. As operational validation of OntoQL, we have developed a complete engine for this language.

## 7. An Engine for OntoQL on OntoDB

To benefit from the advantages of databases (e.g., persistence, scalability), we have chosen to implement *OntoQL* and the *OntoAlgebra* operators on the OntoDB OBDB [9]. This section only describes the implementation of the OntoQL query language on ontology instances. However, a complete implementation of OntoQL, including all its sub-languages, has been developed [40].

### 7.1. *The OntoDB Ontology-Based Database*

Figure 3 presents the OntoDB architecture. It is composed of the following 4 parts.

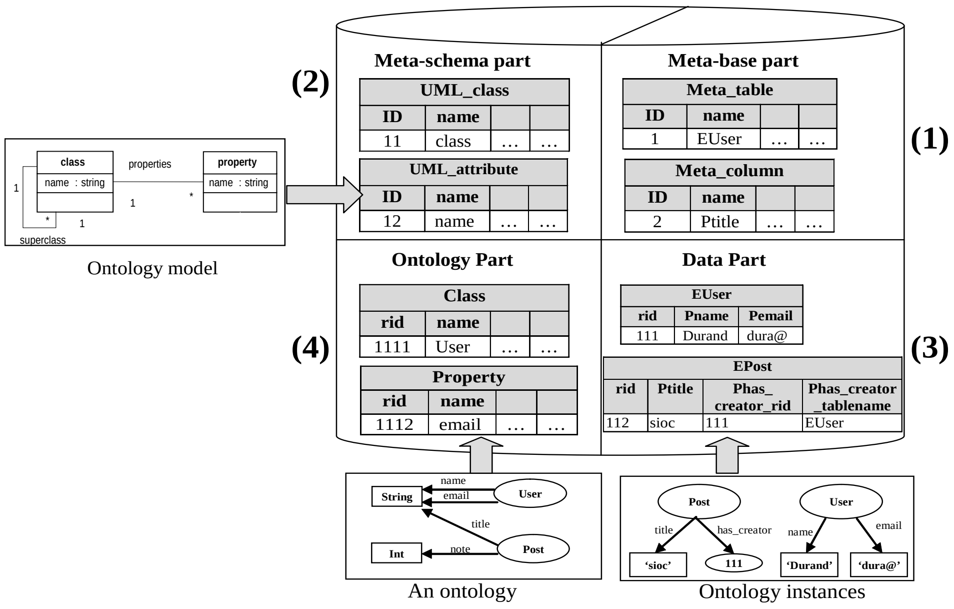


Fig. 3. The OntoDB 4-parts architecture

**- The Meta-Base Part (1)**

The meta-base, also often called catalog system, is a traditional part of any database. It contains system tables used to manage all the data contained in the database. In OntoDB, it contains specifically the description of all the tables and columns defined in the three other parts of the architecture.

**- The Data Part (3)**

This part represents the objects of a domain described by the classes and properties provided by an ontology. These objects are represented following the *horizontal* or *table per class* approach: a table is associated to each concrete class. This table contains the `rid` column to identify instances and one column for each property used to describe instances of the corresponding class. The name of the table (resp. of a column) is the concatenation of `"E"` (resp. `"P"`) with the identifier of the corresponding class (resp. property). This naming convention establishes the link between the ontology part and the data part.

Since a property is represented by a column, a mapping has been established between the ontology model datatypes and the ones of the underlying DBMS: PostgreSQL.

- Primitive datatypes have equivalent datatypes in PostgreSQL.

- The reference type is represented by two columns. The name of the first one is suffixed by `_rid` and provides the `identifier` of the referenced instance. The second one, suffixed by `_tablename` stores the name of the table in which the referenced instance is stored.
- The collection datatype is mapped to the `ARRAY` type of PostgreSQL. Properties whose values are a collection of references are represented by two `ARRAY`-type columns. The first one suffixed by `_rids` stores identifiers of referenced instances. The second one suffixed by `_tablenames` stores the names of the tables in which these instances are stored.

*Example.* Figure 3 part (3) presents an example of the *one table per class* representation for the `User` and `Post` class. The `EUser`[d] table stores instances of the `User` class identified by the `rid` column. We assume that instances of this class are only described by the `name` and `email` string properties and thus, this table has two corresponding `VARCHAR` columns: `Pname` and `Pemail`.

The `EPost` table corresponds to the `Post` class. Its instances are described by the `title` property corresponding to the `VARCHAR` `Ptitle` column. They are also described by the `has_creator` property whose values are references to instances of the `User` class. This property is represented by two columns: one for the identifiers of the referenced instances (`Phas_creator_rid`) and one for the names of the tables which store these instances (`Phas_creator_tablename`).

**- The ontology part (4)**

It contains all the ontologies that define the semantics of the various domains covered by the database. OntoDB supports the PLIB ontology model which includes the core ontology model of OntoQL. Thus, this part contains a set of tables to store PLIB ontologies. As example, we have represented the `Class` and `Property` tables in Figure 3 part (4) which respectively store ontology classes and properties.

**- The meta-schema part (2)**

The meta-schema part records the set up ontology model. For the ontology part, the meta-schema part plays the same role as the one played by the system catalog in traditional DBMS. In Figure 3 part (2), two tables are used to store the entity `class` and its attribute `name` (according to a UML meta-model).

OntoQL statements have been implemented according to this architecture.

### 7.2. *Query Processing of OntoQL on OntoDB*

To process OntoAlgebra operators, each of these operators is interpreted by statements of the underlying query language i.e., SQL. This translation process follows five main steps.

(1) **OntoAlgebra query plan generation.** The query, written in OntoQL, is

[d]For readability, we use names instead of identifiers.

parsed and turned to a tree expression involving OntoAlgebra operators in each node of this tree.

(2) **OntoAlgebra query plan optimization.** We have identified optimization situations (Section 5.3) to reduce the OntoAlgebra query plan. This step is performed together with the previous step to avoid duplication of unnecessary parts of the tree.

(3) **OntoAlgebra query plan transformation into a relational algebra tree.** This translation is achieved by applying a specific set of rules (see below).

(4) **Relational algebra tree optimization.** This step consists in using the different algebraic laws that hold for the relational algebra to turn the relational tree into an optimized equivalent tree.

(5) **Relational algebra tree translation into SQL.** The optimized relational tree is translated into an SQL query according to the underlying DBMS and executed to get the OntoQL query result.

The complete transformation of an OntoAlgebra expression into a relational algebra expression is outside the scope of this paper. We only present in Table 2 two translation rules to convert an OntoAlgebra expression to a relational algebra expression (see [40] for the definition of all rules). In these rules, $\pi$ and $\cup$ represent respectively the projection and union operators of the relation algebra. `C` is a class and $\mathtt{p_1}, \ldots, \mathtt{p_n}$ are the properties defined on this class. Among these properties only $\mathtt{p_1}, \ldots, \mathtt{p_u}$ are used to describe their instances. The datatype of $\mathtt{p_1}$ is a collection of references and the one of $\mathtt{p_2}$ is a single-valued reference. Other properties are primitive.

| | **OntoAlgebra** | **Relational Algebra** |
|---|---|---|
| 1 | $\mathtt{OntoProject}\ (\mathtt{C}, \mathtt{ext}(\mathtt{C}), \{(\mathtt{p_1}, \mathtt{p_1}), \ldots, (\mathtt{p_n}, \mathtt{p_n})\}$ | $\pi_{\mathtt{Pp_1\_rids}, \mathtt{Pp_2\_rid}, \mathtt{Pp_3}, \ldots, \mathtt{Pp_u}, \mathtt{NULL} \rightarrow \mathtt{Pp_{u+1}}, \ldots, \mathtt{NULL} \rightarrow \mathtt{Pp_n}}(\mathtt{EC})$ |
| 2 | $\mathtt{OntoProject}\ (\mathtt{C}, \mathtt{ext}^*(\mathtt{C}), \{(\mathtt{p_1}, \mathtt{p_1}), \ldots, (\mathtt{p_n}, \mathtt{p_n})\}$ | $\mathtt{OntoProject}\ (\mathtt{C}, \mathtt{ext}(\mathtt{C}), \{(\mathtt{p_1}, \mathtt{p_1}), \ldots, (\mathtt{p_n}, \mathtt{p_n})\}) \cup$ $\mathtt{OntoProject}\ (\mathtt{C_1}, \mathtt{ext}^*(\mathtt{C_1}), \{(\mathtt{p_1}, \mathtt{p_1}), \ldots, (\mathtt{p_n}, \mathtt{p_n})\}) \cup$ $\cdots \cup$ $\mathtt{OntoProject}\ (\mathtt{C_n}, \mathtt{ext}^*(\mathtt{C_n}), \{(\mathtt{p_1}, \mathtt{p_1}), \ldots, (\mathtt{p_n}, \mathtt{p_n})\})$ |

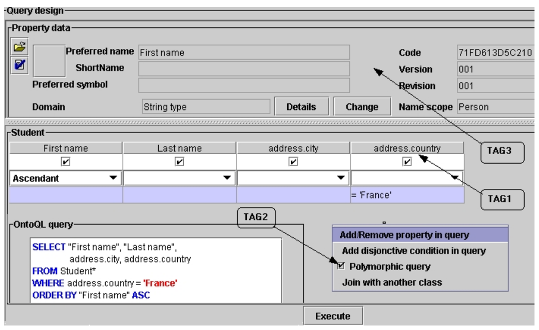
Table 2. Example of OntoAlgebra to relational algebra translation rules

*Semantics.* Rule 1 computes the direct instances of the class `C` with their values for all the properties of this class. The `OntoProject` operator of OntoAlgebra is translated to a projection of the corresponding columns (prefixed by `P`) on the corresponding table (prefixed by `E`). The projection of properties not used to describe instances are interpreted by the projection of the `NULL` values as defined in the OntoAlgebra semantics. The resulting column is renamed (symbol $\rightarrow$) according to the OntoDB naming convention so that other operators are allowed to reference it as a used property.

Rule 2 computes the direct and indirect instances of the class `C`. This recursive operation computes the union of the direct instances of `C` (rule 1) with the deep extent of all its direct subclasses (rule 2).

## 7.3. *Tools to Exploit the OntoQL Engine*

In addition to the implementation of the OntoQL engine, we have developed several tools to ease the exploitation of this language.



| Preferred name | First name | Code | 71FD613D5C210 |
|---|---|---|---|
| ShortName | | Version | 001 |
| Preferred symbol | | Revision | 001 |
| Domain | String type | Name scope | Person |

| First name | Last name | address.city | address.country |
|---|---|---|---|
| ✓ | ✓ | ✓ | ✓ |
| Ascendant | | | |
| | | | = 'France' |

```
SELECT "First name", "Last name",
          address.city, address.country
FROM Student*
WHERE address.country = 'France'
ORDER BY "First name" ASC
```

✓ Polymorphic query

Fig. 4. OntoQBE: a graphical editor for OntoQL queries similar to QBE

*OntoQL*Plus* is an editor of OntoQL instructions similar to SQL*Plus provided by Oracle or *isql* provided by SQLServer. It provides syntax highlighting and a history of the executed commands.

*OntoQBE* is a graphical OntoQL interface provided as a plugin for PLIBEditor which allows users to edit PLIB ontologies. Figure 4 shows the proposed user interface. It extends the QBE interface such as the one provided by *Access* to take into account the object-oriented aspects of the OntoQL language (e.g., path expressions (TAG1) and polymorphism (TAG2)) as well as its ontological aspects (e.g., it shows descriptions of the properties used in the query (TAG3)).

*OntoAPI* is a JAVA representation of the core ontology model of OntoQL. It contains for example the interfaces `EntityClass` and `EntityProperty` representing the classes and the properties. When new elements are added to the core model, this API can be automatically regenerated to take them into account. This API implements the concept of lazy loading proposed for example in the hibernate framework (*http://www.hibernate.org*). With lazy loading, an object is only loaded from the database when an user accesses it through an accessor operator. Thus a class manipulated through the interface `EntityClass` is initially loaded with its primitive attributes (e.g., its name) Then when the `getProperties` method is called, its properties are loaded by calling an OntoQL query.

*JOBDBC* is another API that supports the execution of OntoQL statements from the JAVA programming language. This interface extends the JDBC API. For example the interface `OntoQLResultSet` extends `ResultSet` providing methods to retrieve instances of OntoAPI interfaces as the result of a query. Furthermore the interface `OntoQLResultSetMetaData` extends `ResultSetMetaData` providing methods to get the ontological description (e.g., name in different natural languages, illustration ...) of a column of an `OntoQLResultSet` which references a property.

Source code of these tools is available at `http://www.lias-lab.fr/forge`.

## 7.4. *Application Experiences*

The OntoQL language has been put into practice in various engineering projects (e.g, the Ewok-hub French project[e]) [13, 48, 49]. In particular, its capability to manipulate the ontology model has been extensively used. We have chosen to illustrate two extensions of the ontology model we have realized: (1) annotation of engineering models and (2) user preference handling.

**Engineering Models Annotation**

The CO2 capture and storage rely on various engineering models. Engineers have to face several interpretation difficulties due to the heterogeneity of these models. To ease this process, we have proposed to annotate these models with concepts of ontologies [13]. However, the notions of annotations and engineering models were not available at the metamodel level. Thus, OntoQL introduced these notions as first-order model concepts using a stepwise methodology. First, elements of the engineering models were created with the `CREATE ENTITY` operator. Then, an association table was defined to annotate the engineering models by a class of an ontology. Once the metamodel was extended, OntoQL queries retrieved engineering models from input ontology concepts.

**User Preferences Handling**

When the amount of ontological data (or instances) available becomes huge, queries return an important set of results that must be sorted by a user in order to find the relevant ones. This requirement raised during the eWokHub project where a huge amount of documents and engineering models were annotated by concepts and/or instances of ontologies. As a solution, we have enriched the metamodel to handle user preferences when querying the OBDB. Our proposition is based on a model of user preferences [48] defined at the metamodel level and stored in the OBDB using the `CREATE ENTITY` operator of OntoQL. This preference model is linked to the ontology model by associating preferences to classes or properties of ontologies (`ALTER ENTITY`). Finally, OntoQL has been extended with a `PREFERRING`

[e]`http://www-sop.inria.fr/edelweiss/projects/ewok/`

clause interpreting preferences when querying the OBDB. This interpretation consists in rewriting and expanding the original query by adding additional predicates in its `WHERE` clause.

## 7.5. *Analysis of OntoQL w.r.t our Requirements*

**Requirement 1: Management at the Semantic Level**

OntoQL supports object-oriented operators as well as built-in functions to exploit the semantics of core constructors of ontology-models. An analysis of the expressive power of Semantic Web Query Languages on the set of queries has been reported in [41]. The expression of these queries in OntoQL is available in [40]. This study shows that OntoQL supports the following operators not supported by other studied languages:

- aggregate (`GROUP BY`) and sorting (`ORDER BY`) operators;
- operators on collections (e.g, indexed-access to an element of a collection);
- multilingual operators (e.g, get a value in a given natural language);
- operators on datatypes (e.g., arithmetic operators on integer).

However it does not support recursive queries and specific features of RDF/RDFS: reification, considering everything as a resource, properties without a domain and applying attributes of the ontology model both at the ontology and instance levels.

**Requirement 2: Management of the Different Layers of an Ontology**

Non canonical classes can be defined in OntoQL. Their instances are computed by an SQL view. Notice however that these definitions are not automatic. The user should specify the position of the non canonical class in the hierarchy as well as the query used to compute its instances. Compared to the other view language RVL [32], non canonical classes and canonical classes belong to the same hierarchy. Moreover, if non canonical classes are updated, these modifications are propagated to canonical classes. These differences reflect different points of view on non canonical concepts. RVL separates them to respect the logical data independence. OntoQL considers these two kinds of concepts as part of the same ontology.

The LO layer of an ontology is used in OntoQL to express the same query in different natural languages. Moreover, values of multilingual properties and attributes can be defined in different natural languages. Notice that these capabilities are only available under two constraints: (1) names of classes should be unique for a given namespace and natural language (2) names of properties should be unique for a given class, namespace and natural language. SPARQL does not make these assumptions and thus multilingual querying is not available. However, it introduces useful functions to exploit the LO layer.

**Requirement 3: Generic Ontology Exploitation Language**

OntoQL is based on a core ontology model that can be extended using its ontology definition language (`CREATE ENTITY`). By subsumption the added constructors inherits from the semantics hard-coded in the core ontology model. This capability is useful for many problems as we have seen in 7.4. However, it would also be useful to define a particular semantics for a new constructor (e.g., the computation of the instances of an `owl:UnionOf` classes). This is a work in progress.

**Requirement 4: Full Management of Ontologies and Instances**

OntoQL provides definition, manipulation and query languages at two different levels: ontology and instances. Moreover the two query languages are combined in order to express queries both on ontologies and instances.

**Requirement 5: SQL Compatibility**

Using the namespace mechanism, OntoQL is fully compatible with SQL. Indeed, when no namespaces are specified, SQL queries can be executed using the OntoQL engine. This mechanism also mixes SQL queries used in database applications with ontological queries.

## 8. Conclusion

Several exploitation languages proposals manipulating ontologies and their instances have emerged in the last decade. These languages are often specific to a given ontology model, not fully compatible with the SQL language, and they do not offer a complete exploitation of the different layers (CCO, NCCO and LO) of an ontology. In this paper, we have described the OntoQL language we have designed to address these shortcomings. We have given the complete definition of this language and described some of the many applications both in academia and in industry where this language proved successful.

The OntoQL proposal was built in an incremental design process. As a first step, we have formally defined a data model for ontology and their instances independent of the used ontology model. Indeed, OntoQL manipulates ontologies according to a layered model that characterises different categories of ontologies. This layered model is based on a core ontology model corresponding to the shared constructors of different ontology models. Moreover, OntoQL extends this core ontology model in order to target specific ontology models. As a consequence, the cohabitation of several ontology models and their corresponding ontological data in the same setting becomes possible and thus they can be exploited by the same OntoQL constructs.

As a second step, a formal algebraic semantics was proposed. An algebra of operators, *OntoAlgebra*, for querying ontology and their instances resulted from this definition. It is built by extending the *ENCORE* algebra proposed for object-oriented database. In the same way, the OntoQL language syntax, which semantics is defined by *OntoAlgebra*, was set up by adapting the SQL99 syntax for relational-object databases. As a result, OntoQL proposes different sub-languages to define, manipulate and query ontologies and their instances at different layers. It offers built-in services to access:

- instances at the logical level with a full SQL compatibility;
- instances at the CCO ontological level. OntoQL provides object-oriented operators to query instances from ontologies;
- instances at the NCCO ontological level. Non canonical classes are defined using mechanisms similar to the one defined for views in classical databases;
- instances at the LO ontological level. OntoQL supports the expression of queries in different natural languages;
- ontologies. OntoQL manipulates ontologies defined with its core ontology model. It handles the exploitation of several ontologies in a single setting. This model is extendable using the OntoQL language itself.

Moreover, cross-layers queries can be expressed. Indeed, it is possible to combine SQL queries with ontological queries using the namespace mechanism. This capability is useful to compose existing database applications queries with ontology based queries. As a result, OntoQL encompasses the capability of usual DBMS languages for manipulating data according to their logical model with the capability of Semantic Web ontology languages for manipulating data according to their semantic models given in terms of ontologies. Another characteristic of OntoQL is the expression of queries on both ontologies and their instances. This kind of queries are relevant for two use cases (1) searching ontology concepts with their associated instances and (2) retrieving instances with their ontological descriptions.

As a proof of concepts, we have implemented OntoQL and its algebra on the OntoDB OBDB. This OntoQL engine is equipped with a set of tools similar to the one used for traditional databases (e.g., Query-By-Example or JDBC). These tools have been successfully used in various engineering projects and the source code of this prototype is available at `http://www.lias-lab.fr/forge`.

The development of OntoQL is carried on in several directions. Currently, new mechanisms to extend the core ontology model with new operators and behaviours are under development. Here, the challenge is to enable the automatic definition of operators behaviours without any interactive programming (without end-user programming) but through the exploitation of a metamodel describing behavioural modelling operators and their composition. More precisely, we are working on adding new functions (e.g., Web Services or plugins) that could be triggered during query processing preserving persistence of the models and their instances. Web Services (WS) discovery using OntoQL statements is also another research path currently followed. A registry of semantic WS based on the OntoDB architecture is under design. It captures functional and non functional characteristics of WS using a specific stored metamodel in OntoDB. An extension of the OntoQL language offering Web Services search services in this registry using functional as well as non functional characteristics of WS is under development. Finally, studying the substitutability relationships between semantic Web Services using OntoQL statements is another future research direction.

## References

[1] Gruber, T.R.: A translation approach to portable ontology specifications, in *Knowl. Acquis.* **5**(2) (1993) 199–220.

[2] Bouayad-Agha, N., Casamayor, G., Wanner, L.: Natural Language Generation in the context of the Semantic Web, in the *Semantic Web journal* (2013).

[3] Dragoni, M., da Costa Pereira, C., Tettamanzi, A. G.: A conceptual representation of documents and queries for information retrieval systems by using light ontologies, in *Expert Systems with applications.* **39**(12) (2012) 10376-10388.

[4] Breslin, J. G., O'Sullivan, D., Passant, A., Vasiliu, L.: Semantic Web computing in industry, in *Computers in Industry.* **61**(8) (2010) 729-741.

[5] Wiegand, N.: Ontologies and Database Management System Technology for *The National Map*, in *Cartographica* (2010) 121–126.

[6] Zayas, D. S., Monceaux, A., Ait-Ameur, Y.: Using knowledge and expressions to validate inter-model constraints, In *Proc. of the 18th IFAC Wolrd Congress (IFAC'11)* (2011) 2737-2742.

[7] Bao, J., Kendall E.F., McGuinness, D.L., Patel-Schneider, P.F.: OWL 2 Web Ontology Language, World Wide Web Consortium (2012) `http://www.w3.org/TR/owl2-overview/`.

[8] Pierra, G.: Context Representation in Domain Ontologies and its Use for Semantic Integration of Data, in *Journal Of Data Semantics (JODS)* **X** (2007) 34–43.

[9] Dehainsala, H., Pierra, G., Bellatreche, L.: OntoDB: An Ontology-Based Database for Data Intensive Applications, in *Proc. of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07)* (2007) 497–508.

[10] Wilkinson, K.: Jena Property Table Implementation in *Proc. of the 2nd International Workshop on Scalable Semantic Web Knowledge Base Systems* (2006) 35-46.

[11] Chong, E. I., Das, S., Eadon, G., Srinivasan, J.: An Efficient SQL based RDF Querying Scheme, in *Proc. of the 31st International Conference on Very Large Data Bases (VLDB'05)* (2005) 1216-1227.

[12] Mizoguchi-Shimogori, Y., Murayama, H., Minamino, N.: Class Query Language and its application to ISO13584 Parts Library Standard. In: Proceedings of the 9th European Concurrent Engineering Conference (ECEC'02). (2002) 128–135

[13] Mastella, L.S., Aït-Ameur, Y., Jean, S., Perrin, M., Rainaud, J.F.: Semantic exploitation of persistent metadata in engineering models: application to geological models, in *Proc. of the IEEE International Conference on Research Challenges in Information Science (RCIS 2009)* (2009) 147–156.

[14] Bellatreche, L., Khouri, S., Berkani, N.: Semantic Data Warehouse Design: From ETL to Deployment  la Carte, in *Proc. of the 18th International Conference on Database Systems for Advanced Application (DASFAA'13)* (2013) 64–83.

[15] Romero, A. A., Grau, B. C., Horrocks, I.: MORe: Modular combination of OWL reasoners for ontology classification, in *Proc. of the 11th International Semantic Web Conference (ISWC'12)* (2012) 1-16.

[16] Jonquet, C., LePendu, P., Falconer, S., Coulet, A., Noy, N. F., Musen, M. A., Shah, N. H.: NCBO Resource Index: Ontology-based search and mining of biomedical resources, in *Web Semantics: Science, Services and Agents on the World Wide Web.* **9**(3) (2011) 316–324.

[17] Bailey, J., Bry, F., Furche, T., Schaffert, S.: Semantic Web Query Languages, in *Encyclopedia of Database Systems* (2009) 2583–2586.

[18] Jean, S., Ameur, Y.A., Pierra, G.: Ontology Query Languages for Ontology-Based Databases: a Survey, in *Data Warehousing Design and Advanced Engineering Applications: Methods for Complex Construction* (2009).

[19] Harris, S., Seaborne, A.: SPARQL 1.1 Query Language, W3C Recommendation 21 March 2013 (2013) `http://www.w3.org/TR/sparql11-query/`.

[20] Jean, S., Aït-Ameur, Y., Pierra, G.: Querying Ontology Based Database Using OntoQL (an Ontology Query Language), in *Proc. of On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, OTM Confederated International Conferences (ODBASE'06)* (2006) 704–721.

[21] Jean, S., Aït-Ameur, Y., Pierra, G.: An Object-Oriented Based Algebra for Ontologies and their Instances, in *Proc. of the 11th East European Conference in Advances in Databases and Information Systems (ADBIS'07)* (2007) 141–156.

[22] Jean, S., Aït-Ameur, Y., Pierra, G.: Querying ontology based databases. The OntoQL proposal, in *Proc. of the 18th Internation Conference on Software Engineering and Knowledge Engineering (SEKE'06)* (2006) 166–171.

[23] Jean, S., Pierra, G., Ameur, Y.A.: Domain Ontologies: A Database-Oriented Analysis, in *Web Information Systems and Technologies, International Conferences, WEBIST 2005 and WEBIST 2006. Revised Selected Papers* (2007) 238–254.

[24] Cullot, N., Parent, C., Spaccapietra, S., Vangenot, C.: Ontologies: A Contribution to the DL/DB Debate, in *Proc. of the 1st International Workshop on the Semantic Web and Databases* (SWDB'03) (2003) 109–129.

[25] Brickley, D., Guha, R.: RDF Vocabulary Description Language 1.0: RDF Schema, World Wide Web Consortium (2004) `http://www.w3.org/TR/rdf-schema/`.

[26] Breslin G. J., Harth A., Bojars, U., Descker, S.: Towards Semantically-Interlinked Online Communities, in *Proc. of the 2nd European Semantic Web Conference (ESWC'05)* (2005) 238–254.

[27] Gearon, P., Passant, A., Polleres, A.: SPARQL 1.1 Update, W3C Recommendation 21 March 2013 (2013) `http://www.w3.org/TR/sparql11-update/`.

[28] Bizer, C., Seaborne, A.: D2RQ - Treating Non-RDF Databases as Virtual RDF Graphs, in *Proc. of the 3rd International Semantic Web Conference (ISWC'04)* (2004).

[29] Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: a declarative query language for RDF, in *Proc. of the Eleventh International World Wide Web Conference (WWW'02).* (2002) 592–603.

[30] Cattell, R.G.G.: *The Object Database Standard: ODMG-93*, Morgan Kaufmann (1993).

[31] Magiridou, M., Sahtouris, S., Christophides, V., Koubarakis, M.: RUL: A Declarative Update Language for RDF, in *Proc. of the 4th International Semantic Web Conference (ISWC'05)* (2005) 506–521.

[32] Magkanaraki, A., Tannen, V., Christophides, V., Plexousakis, D.: Viewing the Semantic Web Through RVL Lenses, in *Journal of Web Semantics* **1**(4) (2004) 359–375.

[33] Ziegler, P., Sturm, C., Dittrich, K.R.: Unified Querying of Ontology Languages with the SIRUP Ontology Query API, in *Datenbanksysteme in Business, Technologie und Web (BTW'05)* (2005) 325–344.

[34] Garcia-Alvarado, C., Ordonez, C.: ONTOCUBO: Cube-based Ontology Construction and Exploration, in *Proc. of the 2014 ACM SIGMOD Conference (SIGMOD'14)* (2014) 1083–1086.

[35] Garcia-Alvarado, C., Chen, Z., Ordonez, C.: ONTOCUBE: Efficient Ontology Extraction using OLAP Cubes, in *Proc. of the 20th ACM Conference on Information and Knowledge Management (CIKM'11)* (2011) 2429–2432.

[36] Garcia-Alvarado, C., Ordonez, C.: Query Processing on Cubes Mapped from Ontologies to Dimension Hierarchies, in *Proc. of the 15th ACM International Workshop on Data Warehousing and OLAP (DOLAP'12)* (2012) 57–64.

[37] Sequeda, J. F., Tirmizi S., Corcho O., Miranker D. P.: Survey of directly mapping SQL databases to the Semantic Web, in *Knowledge Eng. Review* **26**(4) (2011) 445–486.

[38] Astrova, I.: Rules for Mapping SQL Relational Databases to OWL Ontologies, in *Proc. of the 2nd International Conference on Metadata and Semantics Research (MTSR'07)* (2007) 415–424.

[39] Zdonik, S.B., Mitchell, G.: ENCORE: An Object-Oriented Approach to Database Modelling and Querying, in *IEEE Data Engineering Bulletin* **14**(2) (1991) 53–57.

[40] Jean, S.: OntoQL, un langage d'exploitation des bases de données à base ontologique, PhD thesis, LISI/ENSMA and University of Poitiers (2007).

[41] Haase, P., Broekstra, J., Eberhart, A., Volz, R.: A Comparison of RDF Query Languages, in *Proc. of the 3nd International Semantic Web Conference (ISWC'04)* (2004) 53–57.

[42] Deutsch, A., Fernández, M. F., Florescu, D., Levy, A. Y., Suciu, D.: A Query Language for XML, in *Computer Networks* **31**(11-16) (1999) 1155–1169.

[43] Codd, E. F.: Relational Completeness of Data Base Sublanguages, in Database Systems, Prentice Hall and IBM Research Report RJ 987 (1972) 65–98.

[44] Özsu, T., Valduriez P.: *Principles of Distributed Database Systems*, Prentice Hall Press (2007).

[45] Eisenberg, A., Melton, J., Kulkarni, K., Michels, J.E., Zemke, F.: SQL:2003 Has Been Published, in *SIGMOD Record* **33**(1) (2004) 119–126.

[46] Bazhar, Y., Chakroun, C., Aït-Ameur, Y., Bellatreche, L., Jean, S.: Extending Ontology-Based Databases with Behavioral Semantics, in *Proc. of On the Move to Meaningful Internet Systems 2012: CoopIS, DOA-SVI, and ODBASE, OTM Confederated International Conferences (ODBASE'12)* (2012) 879–896.

[47] IEC61360-4: Standard data element types with associated classification scheme for electric components - part 4 : Iec reference collection of standard data element types, component classes and terms, Technical report, International Standards Organization (1999).

[48] Tapucu, D., Diallo, G., Aït-Ameur, Y., Ünalir, M.O.: Ontology-based database approach for handling preferences, in *Data Warehousing Design and Advanced Engineering Applications: Methods for Complex Construction* (2009) 248–271.

[49] Belaid, N., Ait-Ameur, Y., Rainaud, J.F.: A semantic handling of geological modeling workflows, in *Proc. of the International ACM Conference on Management of Emergent Digital EcoSystems (MEDES'09)* (2009) 83–90.