A Protégé Plugin for Storing OWL ontologies in Relational Databases

Maria del Mar Roldan-Garcia and Jose F. Aldana-Montes

University of Malaga, Computer Languages and Computing Science Department
Malaga 29071, Spain,
(mmar,jfam)@lcc.uma.es,
WWW home page: http://khaos.uma.es

Abstract. This paper presents a Protégé plugin for storing OWL ontologies in relational databases. It is possible to modify the initial logical and physical storage model according to the needs of the user. In order to do so, the knowledge contained in the ontology is translated to an internal representation, and some configurations files are used. These files define the rules that generate the storage models. Both, the internal representation and the configuration files are XML files. The tool presents a graphical interface. Using this plugin, developers of applications in the Semantic Web will be able to create a relational database for storing their ontologies easily. Furthermore, the storage model can be optimized for the application requirements.

1 Introduction

The OWL language [1] is being widely used to define ontologies in the Web. Its XML based syntax together with its correspondence with Description Logics [2], make it a good candidate to be the standard language for defining ontologies used by Semantic Web applications. However, there are still a few tools that allow us to manipulate, store and query ontologies defined using this language. Description logic based tools, like FaCT [3] or RACER [4] allow us to query simple OWL ontologies, but they do not deal with large amounts of information, and their results can only be applied to very small knowledge bases (with a small number of instances) and these are not the knowledge bases we expect to find in the Semantic Web. Consequently, reasoning algorithms are not scalable and are usually main memory oriented algorithms. Querying and reasoning on instances of ontologies will make the Semantic Web more useful. On the other hand, the database research community has successfully developed a wide theoretical corpus and a mature and efficient technology to deal with large amounts of persistent information. Due to the size of the Semantic Web ontologies (very large ontologies with a very large number of instances), we believe it will be necessary to use database technology in order to provide persistence to the knowledge described by the ontologies, and scalability to the queries and reasoning on this knowledge.
Unfortunately, the application of database technology to reason with instances of OWL ontologies is not a trivial matter. Databases work with a closed-world assumption, while ontology systems apply an open-world assumption. In a database, instances are accepted only if they fully comply with the definitions and constraints stated in the schema, while ontology systems accept instances as long as they do not explicitly contradict the knowledge already in the ontology, without requiring that all expected data be present. This must be taken into account in order to implement the reasoning mechanisms using database technology. Our research objective is to study how several different logical and physical (persistent) storage models influence the performance of querying and reasoning mechanisms. Efficient Storage models are necessary for the future of the Semantic Web, because applications in this area will need to perform complex queries\(^1\) \(^2\) and complex Abox reasoning mechanisms. It is logical to think that applications in the Semantic Web will need to infer new knowledge from the explicit knowledge defined not only in the Tbox (ontology structure) but especially in the Abox (ontology instances). The complex reasoning mechanisms that should be implemented for the Semantic Web applications will need an optimized storage model in order to be efficient and scalable. This storage model will be disk oriented and not main memory oriented in order for it to be scalable. Therefore, we must define knowledge storage methods beyond a simple correspondence with a database logical schema. That is, it is necessary to define both a physical knowledge representation and access paths (indexes) in order to access the knowledge efficiently. We believe this to be an open research issue that should be studied.

To do so, we have developed a tool for storing OWL ontologies in relational databases. In this paper we present this tool. It is built as a plugin for the Protégé\(^3\) environment. Protégé is a free, open source ontology editor and knowledge-base framework. It is based on Java, is extensible through plugins, and provides a foundation for customized knowledge-based application systems. We exploit the features of Protégé in order to manipulate OWL ontologies easily. Using this plugin, developers of Semantic Web Applications can create a database to store their OWL ontologies easily. The tool also creates physical storage structures for the ontology. Developers can modify, delete or create new ones according to the needs of their applications.

\(^1\) Complex queries in the description logic context, i.e. database style queries. Here a query means retrieving instances that satisfy certain (complex) restrictions or qualifications and hence are of interest to a user. This goes beyond conjunctive queries.

\(^2\) Please note that both semantics and evaluation mechanisms for queries in the description logic context are quite different from the corresponding mechanisms in a database context.
2 Related Works

In the past few years there has been a growing interest in the development of systems for storing large amounts of knowledge in the Semantic Web. Firstly, these systems were oriented to RDF storage \[6\] \[7\] \[8\] \[9\]. Nowadays, research is oriented to massive OWL storage. Several alternative approaches using relational technology have been presented, i.e. DLDB \[10\], and Instance Store \[11\].

**DLDB** is a knowledge base that extends a relational database management system (RDBMS) with additional capabilities for making OWL inferences. The main objective of this system is to study how description logic reasoning mechanisms can be combined with an RDBMS, in order to support extensional queries on OWL documents. Ontologies are stored using Microsoft Access as RDBMS. Specifically, the system stores RDF in a relational database. Ontologies are stored creating a table for each class or property definition. The class hierarchy is stored in the system using views. The view of a class is defined recursively, and consists of the union of its table and all the views of its direct sub-classes. Therefore, the view of a class includes the instances of that class plus the inferred instances using the taxonomic reasoning mechanism. The sub-property relationship is stored in a similar way. The system is optimized for medium size ontologies (hundreds of classes and properties). The system provides an API, implemented in java, to query the database. It supports conjunctive queries in a format similar to KIF (http://www.cs.umbc.edu/kse/kif/kif101.shtml). The query is translated to SQL and is sent to the database using JDBC. The query is then evaluated by the RDBMS which returns the results. Reasoning mechanisms are implemented using the FaCT reasoner coupled to the RDBMS. FaCT only supports Tbox reasoning; therefore DLDB only implements those reasoning mechanisms which can be reduced to concept subsumption (concept and property taxonomy reasoning). These reasoning mechanisms are implemented by pre-computing the class/property hierarchy and storing it in the database using views. Using FaCT, we obtain the sub-classes/sub-properties of a given class/property needed to generate the views. The system does not support other OWL reasoning mechanisms.

**Instance Store (IS)** is an approach to a restricted form of Abox reasoning that combines a description logic reasoner with a database. The Instance Store can only deal with free-role Aboxes, i.e. Aboxes that do not contain any axioms asserting role relationships between pairs of individuals. Ontologies are described using OWL. Instance Store only offers persistence to the Abox. Abox assertions are stored in a relational database. An identifier (ids) is assigned to each description (concept) and a table stores individuals and the ids of their associated description. Another table contains description ids and all the primitive concepts in the ontology which subsume them. The primitive concepts which are equivalent to, parent of and child of a given description are also stored in a table. Instance Store provides an API written in Java. This API contains a retrieval method that retrieves all the instances of a given concept. The query is
translated to SQL and is sent to the relational database. Instance Store does not provide Tbox reasoning mechanisms. That is, it is not possible to reason about the structure of the ontology. The only Abox mechanism is, as we said above, instance retrieval.

Our proposal also uses a relational database for storing the OWL ontologies. However, its novelty is that developers can change the storage model according to their applications needs. The plugin also creates a physical model for the ontologies, and this model is also customized. All the information about the storage model is defined by several configuration files. These files can be modified using the plugin in a graphical way.

3 A Protégé Plugin for Storing OWL ontologies

The development of tools for storing and querying ontologies is currently under investigation. The necessity for new advanced reasoning mechanisms that allow us to infer knowledge for use in Semantic Web applications entails the development of efficient storage models in order to implement these reasoning mechanisms efficiently. Reasoning/querying in a knowledge base should be scalable and efficient. We want to fulfill these requirements not in a general context but specifically in the Semantic Web environment. We believe that, in this context, we will find a fairly large number of instances and are convinced that, in the Semantic Web, not only is reasoning on concepts necessary, but also reasoning at the instance level and efficient instance retrieval. We would also like to allow queries and in particular the combination of reasoning and querying procedures. In order to achieve this, we need to study different storage models for OWL ontologies. We believe that a specific physical representation of the ontology instances, and ad hoc index structures will be necessary. Both the representation and the indexes will depend on the expressiveness of the query language and the reasoning implemented. Therefore, the solution will not be unique, but rather a trade-off between expressiveness and efficiency.

In this paper we present a tool for storing OWL ontologies in relational databases. The tool also generates a physical storage model, which can be modified according to the applications requirements. The tool is built as a plugin for the Protégé tool, whose features have been exploited in order to manipulate OWL ontologies easily. Figure 1 shows the architecture of the tool. The storage model generation has two phases. First, the OWL ontology is parsed in order to obtain all the information it contains. This process generates several XML files. Secondly, the user selects the desired storage model, the configuration files are loaded and the database is generated.

Once the database is created, the user can use the graphical interface of the Plugin in order to modify both the logical and the physical model of the database. Figure 2 shows the graphical interface of the plugin. In the left part of the interface, information about the ontology is shown. In the right part, information about the storage model is presented.
Fig. 1. Architecture of the Tool

Fig. 2. Plugin graphical interface
3.1 OWL Ontologies Parser

This component uses the Protégé API to generate four XML documents for each OWL ontology. These XML files are:

- **classes.xml**: Stores all the information about named classes defined by the ontology. This information includes name, superclasses, subclasses and properties.
- **properties.xml**: Stores all the information about properties defined by the ontology. This information includes name, type, domain, range, whether the property is transitive or symmetric and whether the property has an inverse property.
- **restrictions.xml**: Stores all the information about restrictions defined by the ontology. This information includes name, type, restricted class, etc.
- **instances.xml**: Stores all the information about instances defined by the ontology. This information includes name, type, properties, etc.

3.2 Configuration files

All information about the storage model is stored using an internal representation in several configuration files. These are xml files. The plugin allows us to modify the files using the graphical interface. In order to modify the storage model, the Plugin shows the content of the files and the user can modify it.

A storage model is defined by several rules. We define three kinds of rules:

- **Default rules**. By applying these rules a default storage model is generated. These rules must be applied all together.
- **Generic rules**. These rules are applied to the default storage model in order to modify it creating physical storage structures or indexes. They are defined for all elements in the ontology that fulfil the rule antecedent. For example, all classes, all properties, etc.
- **Parametric rules**. These rules also are applied to the default storage model in the same way as generic rules. However, they are defined for a specific element of the ontology, which must be specified before the rule application. The default model does not define parametric rules. These rules will only be used in order to modify the model.

The configurations file contains the implementation of each rule for a specific database management system. This implementation consist of a java method. If a semantic web application developer wants to modify the storage model, he/she should create a new generic or parametric rule and provides its implementation. This is really easy to do using the graphical interface. The new rule will be stored in the corresponding configuration file. Existing rules can be deleted or modified. Figure 3 shows the window to modify an existing generic rule, and figure 4 the window to add a new parametric rule.

Since the new rules are stored, it is possible to define a new storage model by creating or modifying the rules and then storing it. Therefore, a developer
Fig. 3. Modifying a generic rule

Fig. 4. Adding a new parametric rule
could store different models and use them to create his/her databases according to his/her applications requirements.

The tool also provides a component that implements the necessary functions which recover the information about the ontology by accessing the xml files that represent it. For example, the name of the classes or properties defined by the ontology. This component can be used by all developers who want to define new rules.

![EER schema of the database storing the OWL ontologies](image)

Fig. 5. EER schema of the database storing the OWL ontologies

4 A Relational Model for Storing OWL Ontologies

In order to develop our tool, we create a storage model for OWL ontologies in a relational database. In this chapter we will show the storage model (rules) and the configuration files for this particular model.

4.1 Logical Storage Model

Figure 5 shows the entity-relationship model for the database which stores the ontology. The ontology_index entity type has two attributes called seqnum and url, which store the ontology identifier and the ontology name respectively. This entity type is weakly related with a weak entity type called element, which is specialized in three entity types, namely class, properties and instances. For both class and properties there will exist as many specializations as there are classes and properties in the ontology. Therefore, it is possible for each class
and property to determine their instances using the corresponding relationships. Finally, the subclass of relationship allows us to generate the classes hierarchy of the ontology. Table 1 shows the rules created to define the logical model tables.

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Antecedent</th>
<th>Consequent</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule1</td>
<td>ontology</td>
<td>table</td>
<td>creates a table to store the ontology name and id</td>
</tr>
<tr>
<td>rule2</td>
<td>class</td>
<td>table</td>
<td>stores each class in a table</td>
</tr>
<tr>
<td>rule3</td>
<td>instances</td>
<td>table</td>
<td>stores all instances in a table</td>
</tr>
<tr>
<td>rule4</td>
<td>properties names</td>
<td>table</td>
<td>creates a table to store all property names</td>
</tr>
<tr>
<td>rule5</td>
<td>classes names</td>
<td>table</td>
<td>creates a table to store all classes names</td>
</tr>
<tr>
<td>rule6</td>
<td>property</td>
<td>table</td>
<td>stores each property in a table</td>
</tr>
<tr>
<td>rule7</td>
<td>hierarchy</td>
<td>table</td>
<td>creates a table to store a subclass for each class</td>
</tr>
</tbody>
</table>

Table 1. Rules defining the logical model

4.2 Physical Storage Model

The physical storage depends on the needs of the applications. However, our tool implements some rules for defining some indexes. Developers will be able to change these rules or to implement new ones according to their applications requirements. Table 2 shows the rules for creating these indexes.

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Antecedent</th>
<th>Consequent</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule1</td>
<td>restriction</td>
<td>index for the range</td>
<td>creates an index for the object column of each table that represents a restricted property</td>
</tr>
<tr>
<td>rule2</td>
<td>transitive property</td>
<td>index for the range</td>
<td>creates an index for the subject column of each table that represents a transitive property</td>
</tr>
<tr>
<td>rule3</td>
<td>symmetric property</td>
<td>index for the range</td>
<td>creates an index for the subject column of each table that represents a symmetric property</td>
</tr>
<tr>
<td>rule4</td>
<td>hierarchy</td>
<td>index for classes</td>
<td>creates an index for the super column of the hierarchy table</td>
</tr>
<tr>
<td>rule5</td>
<td>class</td>
<td>index</td>
<td>creates an index for the ID column of each table representing a class</td>
</tr>
</tbody>
</table>

Table 2. Rules defining the physical model
4.3 Creating the Storage Model

In order to create this storage model, we have to create the configuration files. This can be made from scratch or using the plugin graphical interface. There are four configuration files for defining a model, one for each kind of rule and another one for their implementations. The default rules are those showed in Table 1. Figure 6 shows the xml code for rule 1. Rules in table 2 are generic rules. The xml code for rule 2 is showed in figure 7. In this example we do not define parametric rules. We will see an example of parametric rules in the next section. Figure 8 shows the SQL code for implementing each rule.

```
<rule>
  <name> rule1 </name>
  <antecedent> ontology </antecedent>
  <consequent> table </consequent>
  <text> creates a table to store the ontology name and id </text>
</rule>
```

**Fig. 6.** XML code for a default rule

```
<rule>
  <name> rule2 </name>
  <antecedent> transitive property </antecedent>
  <consequent> index for the range </consequent>
  <text> creates an index for the subject column of each table that represents a transitive property </text>
</rule>
```

**Fig. 7.** XML code for a generic rule

```
create table S$conceptname$S
  (id varchar(100),
  );

Create index transitiveXX on S$tablename$S (id_dom);
```

-- S$tablename$S is the name of the transitive property

**Fig. 8.** Rules implementation
4.4 Modifying the Storage Model

In order to modify the storage model, developers can define new rules or modify or delete existing ones. Depending on the type of change they want, they should define a generic or a parametric rule. Once the new rule is defined, it is stored in the corresponding configuration file. For each new rule some information has to be provided. This is the name of the rule, the antecedent, the consequent and a text explaining what the rule does. Besides, for each parametric rule the requirement parameters have to be specified. A java file with the rule implementation must also be provided. For example, let us suppose we want to add a new parametric rule that inserts a table in a cluster. The table is a parameter of the rule. Figure 9 shows the xml code of the rule. Note that developers do not have to know the schema of the configurations files, because new rules can be defined using the plugin interface. Figure 10 shows the process to insert this rule using the graphical interface.

```
<rule>
  <name> cluster rule </name>
  <antecedent> table </antecedent>
  <consequent> insert into cluster </consequent>
  <parameter> table_name </parameter>
  <parameter> column_name </parameter>
  <parameter> cluster_name </parameter>
  <text> inserts the table into the cluster using the specified column </text>
</rule>
```

Fig. 9. Parametric rule example

5 Generating a Relational Database for the Univ-bench OWL ontology

In order to show the use of the plugin, in this chapter we present an example that generates a relational database to store the Univ-Bench ontology [12]. This ontology describes universities and departments and the activities that go on in them. It contains a large number of instances which are automatically generated. The OWL ontology is parsed in order to obtain the XML files containing the ontology information (see section 3.1). Using these files, the ontology is shown graphically using the protégé style. Secondly, we select the desired relational model. Note that more than one storage model can exist if we modify the default model and store it as a different one. In our example only one model is available. Once the model is selected, a window asking for a user name and a password for the database is shown (see figure 11). Then, the database is created. In Figure 12 the extension of the database generated for the univ-bench ontology is presented.
Fig. 10. Adding the cluster rule

Fig. 11. Connection to the Oracle database
6 Conclusions and Future Work

In this paper, we present a Protégé plugin for storing OWL ontologies in relational databases. It is possible to modify the initial logical and physical storage model according to the user needs. The tool also presents a graphical interface. Using this plugin, developers of applications in the Semantic Web will be able to create a relational database for storing their ontologies easily. Furthermore, the storage model can be optimized for the applications requirements. An example of how to use the plugin is also presented. In this example, a database to store the univ-bench ontology is created, and some rules for modifying the initial storage model are implemented. In the future step we plan to refine the logical storage model and we will study common query/reasoning patterns (i.e. for any OWL ontology) in order to define an optimized initial physical storage model.

We are currently developing a new plugin that allows us to access the stored ontology using our tool, in order to query and reason with it. This new plugin will provide a graphical interface to create a query/reasoning language and to execute them.

7 Acknowledgements

This work has been supported by the Spanish MEC Grant (TIN2005-09098-C05-01).

References


