A Method for Updating RDB of Ontology while keeping the Synchronization between the OWL and RDB

Ahmed el Orche, Mohamed Bahaj
1, 2 Department of Mathematics & Computer science, Faculty of Science and Technologies, University Hassan 1st, Settat, Morocco

ABSTRACT

Among of reasons for creation of the web ontology is the lack of semantic data stored in relation database (RDB). Ontology may evolve over time, where for some reasons like performance we must keep synchronization between RDB and ontology. This paper proposes an approach to detect any changes applied between two versions of the same ontology, thus updating the RDB schema instead of creating a new schema to reduce redundancy of unchangeable data and reserve a large space. The approach uses a set of SPARQL queries executed on ontology for extract changes between two versions, and then generate SQL queries for lunching in the RDB.

Keywords:
RDB — Relational database
Oi — Version of Ontology i
DT (domain) — Domain of Data Type
DT (name) — Name of Data Type
DT (range) — Range of Data Type
DO (domain) — Domain of Data Object
DO (name) — Name of Data Object
DO (range) — Range of Data Object
C (name) — Name of class
DTik — Data type (k) in version of ontology i
DOik — Data object (k) in version of ontology i
Column (name) — Name of Column
Column (type) — Type of Column

1. INTRODUCTION

For some reasons such as performance and development of the web semantic, we must keep synchronization between ontology and associated database. That need every time an update of database or ontology after any improvement.

Christian et al. [1] have introduced the notion of "Relational.OWL" which consists to represent a RDB into RDF/OWL, and then use the SPARQL query language to make queries on this representation in order to create an equivalent ontology. [2] presents an approach of automatic conversion from RDB to Ontology on two levels, the first one its purpose to extract the RDB schema and convert it directly to an ontology model (TBOX), the subject of the second level is populate the ontology by individuals (ABOX) using data of different records of the RDB and basing on the model of the ontology. [3] Investigate possibilities of storing multi-version ontology’s in RDB which reduces the redundancy of storage of unchanged ontology components. [6] is designed to speed up ontology querying, which is supposed to represent the most frequent and therefore critical usage of ontology’s. Bottom-up change detection method for a database or metadata perspective is proposed in [7] to preserve compatibility serialization between ontology and the associated database.

This paper presents an approach for keeping the synchronization between an ontology and RDB, we will extract all changes of ontology thought SPARQL queries, interpret these changes to SQL queries for execute it into RDB. Thus we can reduce the redundancy of storage ontology components and reserve a large space of the RDB after synchronization operation.

For represent more RDB objects into associated ontology, to enrich it and to check the updating of data in RDB, we propose an implementation of RDB constraints in ontology with some examples.

2. PRELIMINARY AND DEFINITIONS

[2] Defines RDB as a set of tables. Each table is characterized by its name, a list of its attributes and a list of its records. Each attribute is characterized by its name, its type, a Boolean to test whether the field is a primary key, another Boolean to test whether the converted field is a foreign key, and the table to which it refers if it is a foreign key. Record is a set of pairs (attribute, value), which associates for each attribute a corresponding value.

The ontology taken in our approach is O = TBOX ∪ ABOX, where TBOX is the model and ABOX is the list of individuals.
TBOX is a model defined as a set of classes, a set of data type and a set of data object, each class is characterized by name. Each of data type and data object is characterized by its name, its domain and its range as indicated below:

\[
\text{TBOX} := \{C, \{DT\}, \{DO\}\} \\
C = \{\text{Name}, \text{Class Parent}\}, \hspace{1cm} \text{DT} = \{\text{DT}\text{1}, \text{DT}\text{2}, \ldots, \text{DT}\text{im}\} \hspace{1cm} \text{DO} = \{\text{DO}\text{1}, \text{DO}\text{2}, \ldots, \text{DO}\text{ip}\} \\
\]

**Definition 1:**
An OWL ontology is viewed as four-tuple
\[
O = <C, \text{DT}, \text{DO}, I> \\
\]
Where:
\[
C = \{C_1, C_2, \ldots, C_n\} \text{ is a set of classes} \\
\text{DT} = \{\text{DT}\text{1}, \text{DT}\text{2}, \ldots, \text{DT}\text{im}\} \text{ is a set of data type} \hspace{1cm} \text{DO} = \{\text{DO}\text{1}, \text{DO}\text{2}, \ldots, \text{DO}\text{ip}\} \text{ is a set of data object} \hspace{1cm} I = \{I_1, I_2, \ldots, I_n\} \text{ is a set of individuals} \\
\]

**Definition 2:**
A data type of class is viewed as three-tuple
\[
\text{DT} = \langle \text{DT}\text{1}, \text{DT}\text{2}, \text{DT}\text{3}\rangle \\
\]

**Definition 3:**
A data object of class is viewed as three-tuple
\[
\text{DO} = \langle \text{DO}\text{1}, \text{DO}\text{2}, \text{DO}\text{3}\rangle \\
\]

**Definition 4:**
An OWL ontology version is viewed as four-tuple
\[
O_i = <C_i, \text{DT}_i, \text{DO}_i, I_i> \\
\]
Where:
\[
C_i = \{C_1, C_2, \ldots, C_{im}\} \text{ is a set of classes}, \hspace{1cm} \text{DT}_i = \{\text{DT}\text{1}_i, \text{DT}\text{2}_i, \ldots, \text{DT}\text{im}_i\} \text{ is a set of data type}, \hspace{1cm} \text{DO}_i = \{\text{DO}\text{1}_i, \text{DO}\text{2}_i, \ldots, \text{DO}\text{ip}_i\} \text{ is a set of data object}, \hspace{1cm} I_i = \{I_{1i}, I_{2i}, \ldots, I_{ni}\} \text{ is a set of individuals.} \\
\]

**Rule:**
Given an ontology \(O_1\), \(O_2\), and \(O_j\) are two versions of the same ontology where \(i < j\)
\[
\text{a. } C_k \notin C_j \Rightarrow \text{ change found in ontology.} \\
\text{b. } C_k \notin C_i \Rightarrow \text{ change found in ontology.} \\
\text{c. } C_k \text{ is a class in ontology versions } O_i \text{ and } O_j \\
\]

where \(i < j\):
\[
\text{DT}_i = \{\text{DT}\text{1}_i, \text{DT}\text{2}_i, \ldots, \text{DT}\text{im}_i\} \text{ is a set of data type of } O_i \text{ where } \text{DT}_i(\text{domain}) = C_k \hspace{1cm} \text{DT}_j = \{\text{DT}\text{1}_j, \text{DT}\text{2}_j, \ldots, \text{DT}\text{im}_j\} \text{ is a set of data type of } O_j \text{ where } \text{DT}_j(\text{domain}) = C_k \\
\]

\text{F} is function from a set \(\text{DT}_i\) to a set \(\text{DT}_j\)
\[
\text{F}: \text{DT}_i \rightarrow \text{DT}_j \\
<\text{DT}_k(\text{domain}), \text{DT}_k(\text{range})> \rightarrow <\text{DT}_k(\text{domain}), \text{DT}_k(\text{range})> \\
\]

If \(F\) is not bijective function then change found in ontology.

\[
\text{DT}_k(\text{range}) \rightarrow \text{DT}_k(\text{range}) \\
\]

If \(F\) is not bijective function then change found in ontology.
3. UPDATE RDB MODEL

The figure above describes a migration from RDB to ontology (version 1). The same ontology passes from version 1 to version 2 after some improvements. Then we will propose an approach able to detect all changes between two ontology versions and create SQL queries to update RDB.

3.1 Constraints Implementation

In any RDB there are a various type of constraints like primary keys, foreign keys, unique constraints, not null constraints, and check constraints, but in [2] there are not implementations for constraints in the ontology. We propose bellow to enrich the ontology thought RDB constraints.

Every constraint will have an ID unique, a domain for specify the table of the constraint and a range for specify the column of the constraints.

Obviously the ID of the constraint must be the same ID as in the database, but in followed we propose some rules to follow if the ontology amended by adding new constraints.

### 3.1.1 PK constraint

The ID of primary key constraint will prefixed by “PK” followed by the name of table, the domain will be the name of table and the range will be the name of column.

For any primary key having multiple columns, we have taken an implementation with the same domain, same ID and different range for each column.

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Primary Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>PK_TableName</td>
</tr>
<tr>
<td>Domain</td>
<td>Table name</td>
</tr>
<tr>
<td>Range 1</td>
<td>(Column_1_name ...)</td>
</tr>
<tr>
<td>Range N</td>
<td>Column_N_name) of primary key</td>
</tr>
</tbody>
</table>

**OWL implementation for Primary key**

```xml
<owl:ObjectProperty rdf:ObjectPropertyType ="PK" rdf:ID="PK_Table">  
<rdfs:domain rdf:resource="#Table"/>  
<rdfs:range rdf:resource="# Column 1"/> 
</owl:ObjectProperty>
```

**Example**

```xml
owl:ObjectProperty rdf:ObjectPropertyType ="PK" rdf:ID="PK_Module">  
<rdfs:domain rdf:resource="#Module"/>  
<rdfs:range rdf:resource="# idModule "/> 
</owl:ObjectProperty>
```

### 3.1.2 FK constraint

The ID of foreign key constraint will prefixed by “FK” followed by the name of table, followed by the name of the referenced table, the domain will be the name of table followed by the foreign key column, and the range will be the referenced table followed by the name of referenced column.

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Foreign Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>FK_TableName</td>
</tr>
<tr>
<td>Domain</td>
<td>Table name.column of foreign key</td>
</tr>
<tr>
<td>Range</td>
<td>Referenced table.Referenced Column</td>
</tr>
</tbody>
</table>
OWL implementation for foreign key

Example

3.1.3 UNIQUE Constraint
The ID of UNIQUE constraint will prefixed by “U” followed by the name of table, followed by sequence of this constraint, the domain will be the name of table, and the range will be the column of constraint.

For UNIQUE constraint with multiple columns we have taken multiple implementations with the same domain, ID, and different range for each column.

Table 3: OWL to RDB mapping for UNIQUE constraint

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>UNIQUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>U_TableName_sequence</td>
</tr>
<tr>
<td>Domain</td>
<td>Table name</td>
</tr>
<tr>
<td>Range1</td>
<td>(Column_1_name of UNIQUE constraint)</td>
</tr>
<tr>
<td>RangeN</td>
<td>(Column_N_name of UNIQUE constraint)</td>
</tr>
</tbody>
</table>

OWL implementation for UNIQUE constraint

Example

3.1.4 NOT NULL Constraint
The ID of NOT NULL constraint will prefixed by “NN” followed by the name of table, followed by sequence of this constraint, the domain will be the name of table, and the range will be the column of constraint.

Table 4: OWL to RDB mapping for NOT NULL constraint

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>NOT NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>NN_TableName_sequence</td>
</tr>
<tr>
<td>Domain</td>
<td>Table name</td>
</tr>
<tr>
<td>Range</td>
<td>Column of constraint</td>
</tr>
</tbody>
</table>

OWL implementation for NOT NULL constraint

Example

3.1.5 CHECK Constraint
The ID of CHECK constraint will prefixed by “CHK” followed by the name of table, followed by sequence of this constraint, the domain will be the name of table. In this constraint we propose add two new attributes for the range, source for contain the column name and value for present the constraint value.

Table 5: OWL to RDB mapping for Check constraint

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>CHECK</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>CHK_TableName_sequence</td>
</tr>
<tr>
<td>Domain</td>
<td>Table name</td>
</tr>
<tr>
<td>Range.Source</td>
<td>Column of constraint</td>
</tr>
<tr>
<td>Range.Value</td>
<td>Value of constraint</td>
</tr>
</tbody>
</table>
### OWL implementation for CHECK constraint

```
<owl:ObjectProperty rdf:ID="CHK_TableName_seq">
<rdfs:domain rdf:resource="#TableName"/>
<rdfs:range rdf:source="# Column" rdf:value="# ConstraintSource"/>
</owl:ObjectProperty>
```

### Example

- <owl:ObjectProperty rdf:ID="CHK_Teach_01">
  - <rdfs:range rdf:source="# semester " rdf:value="# semester in('S1','S2','S3','S4','S5','S6') />
  - <owl:ObjectProperty rdf:ID="CHK_TableName_seq " />
</owl:ObjectProperty>

### 3.2 Generate SQL Queries

For updating the RDB schema we must checking all rules below for generate SQL queries to execute in the RDB.

We use SPARQL queries to extract all changes between two ontology versions.

**Steps to build SQL queries:**

a. `Table(name) <-> C(name)`
b. `Column (name) <-> DT(name)` where `DT (domain) = C (name)`
c. `Column(type) <-> DT(range)` where `DT (domain) = C (name)` and `DT = <Table (name), Column (name), DT (range)>`
d. `Constraint(name) <-> DO(name)` where `DO (domain) = C (name)`
e. `Constraint(value) <-> DO(range)` where `DO (domain) = C (name)` and `DO = < Table (name), Constraint (name), DO (range)>`

**Drop table:**

If `Cik ∈ Oi` and `Cjk ∈ Oj`
Then the table associated to `Cik` will be deleted from RDB:
```
DROP TABLE "Table (name)"
```

The query will be executed is:

**Add table:**

If `Cik ⊈ Ci` and `Cik ⊈ Cj`
Then a new table must be associated to `Cik`:
```
DROP TABLE "Table (name)"
```

The query will be executed is:

**Alter table:**

If `Cik ∈ Oi` and `Cjk ∈ Oj`
Then new column associated to `Cik` will be dropped from table:
```
ALTER TABLE "Table (name)" DROP COLUMN "Column (name)"
```

The query will be executed is:

**If `Cik ⊈ Oi` and `Cjk ⊈ Oj}` then the column associated to `Cik` will be added to table:
```
ALTER TABLE "Table (name)" ADD COLUMN "Column (name)" ...
```

The query will be executed is:

**For each `DT j` in `DT i` where `DT j (domain) = Cj(name)`:
```
Column p (name) <-> DT p (range);
Column p (type) <-> DT p (range);
where DTjp (name) = Column p (name)
```

The query will be executed is:

**If `DT j ⊈ DT i` then the table associated to `DT j` will be modified:**
```
Table (name) <-> Ck (name);
Column (name) <-> DT jk (name); where DT jk (domain) = Ck
```

The query will be executed is:

**If `Cik ⊈ Oi and `Cik ⊈ Oj}` then the column associated to `DT jk` will be dropped from table:
```
ALTER TABLE "Table (name)" DROP COLUMN "Column (name)"
```

The query will be executed is:

**If `Cik ⊈ Oi` and `Cik ⊈ Oj}` then the column associated to `DT jk` will be added to table:
```
ALTER TABLE "Table (name)" ADD COLUMN "Column (name)"
```

The query will be executed is:
For each $i < j;
\text{DO}_i = \{ \text{DO}_{i1}, \text{DO}_{i2} \ldots \text{DO}_{in} \} \text{ is a set of data type of } O_i \text{ and } \text{DO}_{i}(\text{domain}) = C_k$
\text{DO}_j = \{ \text{DO}_{j1}, \text{DO}_{j2} \ldots \text{DO}_{jm} \} \text{ is a set of data type of } O_j \text{ and } \text{DO}_{j}(\text{domain}) = C_k$

if $\text{DO}_k \notin \text{DO}_i$ Then the constraint corresponding to $\text{DO}_k$ will be deleted from RDB:

Table_name \leftarrow C_k(name);
Constraint(name) \leftarrow \text{DO}_k(name);

The query will be executed is:

```
ALTER TABLE "Table_name" DROP CONSTRAINT "Constraint(name)";
```

if $\text{DO}_k \in \text{DO}_i$ Then new constraint corresponding to $\text{DO}_k$ will be added to RDB:

Table_name \leftarrow C_k(name);
Constraint(name) \leftarrow \text{DO}_k(name); \text{ where } \text{DO}_k(\text{domain}) = C_k
Constraint(\text{column+value}) \leftarrow \text{DO}_k(\text{range}); \text{where}
\text{DO}_k(\text{domain}) = C_k And
\text{DO}_k = < \text{Table_name}; \text{Constraint(name)}; \text{DO}_k(\text{range})>

if $\text{DO}_k \in \text{DO}_i$ and $\text{DO}_k = < C_k, \text{DO}_k(name); \text{DO}_k(\text{range})>$
and \{ $\text{DO}_k(name) = \text{DO}_k(name)$
and $\text{DO}_k(\text{range}) \neq \text{DO}_k(\text{range})$ \} then the constraint corresponding to $\text{DO}_k$ will be modified:

Table_name \leftarrow C_k(name);
Constraint(name) \leftarrow \text{DO}_k(name); \text{ where } \text{DO}_k(\text{domain}) = C_k
Constraint(\text{column+value}) \leftarrow \text{DO}_k(\text{range}); \text{where}
\text{DO}_k(\text{domain}) = C_k And
\text{DO}_k = < \text{Table_name}; \text{Constraint(name)}; \text{DO}_k(\text{range})>

### 3.3 Algorithms

Load (ontology $O_k$, ontology $O_j$)

Create CDFK list of classes in $O_i$ and not in $O_j$ with FK constraints
For each $C$ of CDFK do

Table_name \leftarrow C(name)
Drop table "Table_name"
End for each

Create CDNFK list of classes in $O_i$ and not in $O_j$ without FK constraints
For each $C$ of CDNFK do

Table_name \leftarrow C(name)
Drop table "Table_name"
End for each

Create CI list of classes in $O_j$ and not in $O_i$
For each $C$ of CI do

Table_name \leftarrow C(name)
Create ATR list <C(name), DT(name), DT(range)>
For each Attribute of ATR do

Column_name \leftarrow DT(name)
Column_type \leftarrow DT(range)
End for each
Create table “Table_name”
End for each

Create CM list of classes in $O_i$ and in $O_j$
For each $C$ of CM do

Create DTI list of <C(name), DT(name), DT(range)> of $O_i$
Create DTJ list of <C(name), DT(name), DT(range)> of $O_j$
Create DTK list in {DTI,DTJ}

When Table_name populated and without any constraint
Drop table “Table_name”
For each element of DTJ do

Column_name[element] \leftarrow DT(name)
Column_type[element] \leftarrow DT(range)
End for each
Create Table Table_name
Import data from temporary table

Table_name to Table_name
Drop temporary table “Table_name_tmp”
Else if Table_name is not populated
Then
For each element in DTK

Column_name \leftarrow DT(name)
Alter table “Table_name”
Add column “Column_name”
End if;
End if;
Create DTK list in {DTJ,DTI}
For each element in DTK do

Column_name \leftarrow DT(name)
Column_type \leftarrow DT(range)
Alter table “Table_name”
Modify column “Column_name(column_range)”
End for each;
Create DTK list in [DT / DT in DTJ and DT in DTJ and DTJ(name) = DTJ(name) and DTJ(range) ≠ DTJ(range) ]
For each element in DTK do

Column_name \leftarrow DTJ(name)
Column_type \leftarrow DTJ(range)
Alter table “Table_name”
End for each;
Create DOI list of <C(name), DO(name), DO(range)> of $O_i$
Create DOJ list of <C(name),DO(name),DO(range)> of O.
Create DOK list in [DOI,DOJ].
For each element in DOI do
  Constraint_name <- DO(name)
  After table “table_name”: Drop constraint “constraint_name”;
End for each;
Create DOK list in [DOI,DOJ].
For each element in DOI do
  Constraint_name <- DO(name)
  Constraint_type <- DO(range)
  After table “table_name”: Add constraint “constraint_name”;
End for each;
Create DOK list in [DO / DO ∈ DOI and DT ∈ DOJ and DO_i(name) = DO_j(name) and DO_i(range) = DO_j(range) ]
For each element in DOI do
  Constraint_name <- DO(name)
  Constraint_type <- DO(range)
  After table “table_name”: Modify constraint “constraint_name”;
End for each;
End for each;

4. CONCLUSION AND PERSPECTIVE
In this paper we proposed rules and approach for detect any change between two ontology versions to synchronize the last version with associated RDB, for that we used just SPARQL queries for extract all changes then create the corresponding SQL queries to execute in RDB. We added RDB constraints in the ontology to check the data before updating. But we did not address data synchronization. So in the next work we will extract the changed ontology data to update RDB.

REFERENCES

AUTHOR PROFILES
Mohamed BAHAJ He got his PhD in Applied Mathematics, from University of Pau, France, in 1993. He is now working as a Professor at the Department of Mathematics & Computer Sciences, University of Hassan 1st, Faculty of Sciences & Technologies Settat, Morocco. His research interests include pattern recognition, Semantic web & Ontology in MAS, Controls of mobiles agents.
Ahmed EL ORCHE He is Phd researcher in Department of Mathematics & Computer Sciences, University of Hassan 1st, Faculty of Sciences & Technologies of Settat, Morocco. His research interests to Migration databases to web ontology using query languages.