

Incremental Maintenance of RDF Views of Relational Data

Vania Maria P. Vidal¹, Marco Antonio Casanova², Diego Sá Cardoso¹

¹Federal University of Ceará, Fortaleza, CE, Brazil
{vvidal, dcardoso}@lia.ufc.br

²Department of Informatics – Pontifical Catholic University of Rio de Janeiro, RJ, Brazil
casanova@inf.puc-rio.br

Abstract. This paper proposes an incremental maintenance strategy, based on rules, for RDF views defined on top of relational data. The first step relies on the designer to specify a mapping between the relational schema and a target ontology and results in a specification of how to represent relational schema concepts in terms of RDF classes and properties of the designer's choice. Using the mappings obtained in the first step, the second step automatically generates the rules required for the incremental maintenance of the view.

Keywords: RDF View, Incremental View Maintenance, Correspondence Assertions.

1 Introduction

As RDF becomes the facto standard for publishing structured data over the Web and since most business data is currently stored in relational database systems, the problem of publishing relational data in RDF format has special significance. A general and flexible way to publish relational data in RDF format is to create RDF views of the underlying relational data. The contents of views can be materialized to improve query performance and data availability. However, to be useful, a materialized view must be continuously maintained to reflect dynamic source updates. Basically, there are two strategies for materialized view maintenance. *Re-materialization* re-computes view data at pre-established times, whereas *incremental maintenance* periodically modifies part of the view data to reflect updates to the database. It has been shown that incremental maintenance generally outperform full view re-computation [1,2,6,9,13,17,18,20].

This paper proposes an incremental maintenance strategy, based on rules, for RDF views defined on top of relational data. The strategy has two major steps: The *mapping generation step* relies on the designer to specify a mapping between the relational schema and a target ontology and results in a specification of how to represent relational schema concepts in terms of RDF classes and properties of the designer's choice. The mapping induces a RDF view that is exported from the data source. The *view maintenance rules generation step* automatically generates the rules required for incremental maintenance of the RDF view from the mappings.

Our solution has the following major points. First, we propose correspondence assertions as a convenient way to specify customized mappings between target RDF vocabularies and base relational schemas. The benefits of using declarative formalisms for schema mappings are well known [3,12]. The concept of correspondence assertions was used in an earlier paper [20] to investigate XML views, but it proved to be much simpler to apply the concept to the context discussed in this paper. It is important to pointing out that the problem of generating schema mappings is outside the scope of this paper.

Second, the views that we address are focused on schema-directed RDF publishing. As such, the correspondence assertions induce schema mappings defined by the class of projection-selection-equi-join queries, which supports most types of data restructuring that are common in data publishing. We make a compromise in constraining the expressiveness of mappings to obtain an algorithm that is very efficient. Furthermore, the views are self-maintainable.

Third, our rules can be implemented using triggers. Hence, no middleware system is required, since triggers are responsible for directly propagating the changes to the materialized views.

Fourth, most of the work is done at view definition time. Based on the mappings, at view definition time, we are able to: (i) identify all source updates that are relevant to the view; and (ii) define the view maintenance statements required to maintain the view w.r.t a given relevant update. We emphasize that the view maintenance statements are defined based solely on the source update and current source state and, hence, no access to the materialized view is required. This is important when the view is maintained externally [18], because accessing a remote data source may be too slow.

Lastly, the view maintenance statements propagated by the rules do not require any additional queries over data source to maintain the view. Again, this becomes important when the view is maintained externally [18].

The use of rules is therefore a very effective solution for incremental maintenance of external views. However, creating rules that correctly maintain an RDF view can be a complex process, which calls for tools that automate the rule generation process. In this paper, we show that, based on a set of correspondence assertions [20], one can generate, automatic and efficiently, all the rules required to maintain a materialized view. Our formalism allows us to justify that the rules generated by our approach correctly maintain the view.

The remainder of this paper is organized as follows. Section 2 summarizes related work in the area of incremental view maintenance. Section 3 introduces the correspondence assertions. Section 4 presents the example used throughout the paper. Section 5 describes our approach for the automatic generation of incremental view maintenance rules, based on the correspondence assertions. Section 6 contains the conclusions.

2 Related Work

The problem of Incremental View Maintenance has been extensively studied for relational view [6, 11] as well as for object-oriented view [2, 13]. There have been also

incremental maintenance algorithms for semi-structured views [1, 14, 21] and XML views [7, 9, 17, 20]. Different data models and view specification languages have been assumed by a number of researchers. The algorithms in [1, 14, 21] are developed for views defined with a query over graph structures. The views considered in [7, 9] are defined using an XML algebra over XML trees, and the views in [17] are defined using path expressions over XML documents.

The incremental algorithm in [4] maintains XML documents produced by an ATG, a formalism for mapping a relational schema to a predefined (possibly recursive) DTD. In their approach, a middleware system interacts with the underlying DBMS and maintains a hash index and a sub tree pool for the external XML view. The main problem with this approach, not to mention the high complexity of the algorithm, is that it requires several round-trips between the middleware and the DBMS. Therefore, the view is not self maintainable, which is a desirable feature for external views (view stored outside the DBMS). Other drawbacks are that the use of in-memory hash table limits the technique for large documents cached in a middleware, and it is not possible to detect irrelevant updates.

The algorithm in [20] incrementally maintains materialized XML views of relational data, in the context of the SQL/XML [8] standard. The algorithm has four major steps: first, it identifies the view paths that are relevant to a base update; second, it identifies all elements in a relevant path that are affected by the base update; third, it generates the list of updates required to maintain the affected elements; and, finally, it sends the list of updates to the view.

None of the above techniques can be directly applied to RDF views of relational data.

3 Correspondence Assertions

3.1 Basic Concepts and Notation

As usual, we denote a *relation scheme* as $R[A_1, \dots, A_n]$ and adopt *mandatory* (or *not null*) *attributes*, *keys*, *primary keys* and *foreign keys* as relational constraints. In particular, we use $F(R:L, S:K)$ to denote a foreign key, named F , where L and K are lists of attributes from R and S , respectively, with the same length. We also say that F *relates* R and S .

A *relational schema* is a pair $S=(\mathbf{R}, \mathbf{\Omega})$, where \mathbf{R} is a set of relation schemes and $\mathbf{\Omega}$ is a set of relational constraints such that: (i) $\mathbf{\Omega}$ has a unique primary key for each relation scheme in \mathbf{R} ; (ii) $\mathbf{\Omega}$ has a mandatory attribute constraint for each attribute which is part of a key or primary key; (iii) if $\mathbf{\Omega}$ has a foreign key of the form $F(R:L, S:K)$, then $\mathbf{\Omega}$ also has a constraint indicating that K is the primary key of S . The *vocabulary* of S is the set of relation names, attribute names, etc. used in S . Given a relation scheme $R[A_1, \dots, A_n]$ and a *tuple variable* t over R , we use $t.A_k$ to denote the *projection* of t over A_k . We use *selections* over relation schemes, defined as usual.

Let $S=(\mathbf{R}, \mathbf{\Omega})$ be a relational schema and R and T be relation schemes of S . A list $\varphi=[F_1, \dots, F_{n-1}]$ of foreign key names of S is a *path from* R *to* T iff there is a list R_1, \dots, R_n of relation schemes of S such that $R_1=R$, $R_n=T$ and F_i *relates* R_i and R_{i+1} . We say that tuples of R *reference tuples of* T *through* φ . A path φ is an *association path* iff

$\varphi=[F_1,F_2]$, where the foreign keys are of the forms $F_1(R_2:L_2,R_1:K_1)$ and $F_2(R_2:M_2,R_3:K_3)$. For example, consider the relational schema *ISWC_REL* in Figure 1. The list of foreign keys names $[Fk_Publications,Fk_Authors]$ is an association path from *Papers* to *Persons*, but $[Fk_Publications,Fk_Persons]$ is not even a path.

We also recall a minimum set of concepts related to ontologies. A *vocabulary* V is a set of *classes*, *object properties* and *datatype properties*. An *ontology* is a pair $O=(V,\Sigma)$ such that V is a vocabulary and Σ is a finite set of formulae in V , the *constraints* of O . Among the constraints, we consider those that define the *domain* and *range* of a property, as well as *cardinality constraints*, defined in the usual way.

3.2 Definition of the Correspondence Assertions

This section introduces the notion of correspondence assertion, leaving examples to Section 3. Let $S=(R,\Omega)$ be a relational schema and $O=(V,\Sigma)$ be an ontology and assume that Σ has constraints defining the domain and range of each property.

Definition 3.1: A *class correspondence assertion (CCA)* is an expression of one of following forms:

- (i) $\Psi: C \equiv R[A_1, \dots, A_n]$
- (ii) $\Psi: C \equiv R[A_1, \dots, A_n] \sigma$

where Ψ is the *name* of the CCA, C is a class of V , R is a relation name of S , A_1, \dots, A_n are the attributes of the primary key of R , and σ is a selection over R . We also say that Ψ *matches* C with R .

Definition 3.2: An *object property correspondence assertion (OCA)* is an expression of one of following forms:

- (i) $\Psi: P \equiv R$
- (ii) $\Psi: P \equiv R / \varphi$

where Ψ is the *name* of the OCA, P is an object property of V and φ is a path from R .

Definition 3.3: A *datatype property correspondence assertion (DCA)* is an expression of one of following forms:

- (i) $\Psi: P \equiv R / A$
- (ii) $\Psi: P \equiv R / \{A_1, \dots, A_n\}$
- (iii) $\Psi: P \equiv R / \varphi / B$
- (iv) $\Psi: P \equiv R / \varphi / \{B_1, \dots, B_n\}$

where Ψ is the name of the DCA, P is a datatype property of V , R is a relation name of S , A is an attribute of R , A_1, \dots, A_n are attributes of R , φ is a path from R to T , B is an attribute of T , and B_1, \dots, B_n are attributes of T .

Definition 3.4: A correspondence assertion is *relevant to a relation* R iff it is of one of the following forms:

- (i) $\Psi: C \equiv R[A_1, \dots, A_n]$
- (ii) $\Psi: C \equiv R[A_1, \dots, A_n] \sigma$
- (iii) $\Psi: P \equiv R$
- (iv) $\Psi: P \equiv R / \varphi$

- (v) $\Psi: P \equiv R / A$
- (vi) $\Psi: P \equiv R / \{A_1, \dots, A_n\}$
- (vii) $\Psi: P \equiv R / \varphi / B$
- (viii) $\Psi: P \equiv R' / \varphi$ where φ has a foreign key of R .
- (ix) $\Psi: P \equiv R' / \varphi / \{B_1, \dots, B_n\}$ φ has a foreign key of R .
- (x) $\Psi: P \equiv R' / \varphi / B$, φ has a foreign key of R .

Definition 3.5: A *mapping* between V and S is a set A of correspondence assertions such that:

- (i) If A has an OCA of the form $P \equiv R$, then A must have a CCA that matches the domain of P with R , and a CCA that matches the range of P also with R .
- (ii) If A has an OCA of the form $P \equiv R/\varphi$, where φ is a path from R to T , then A must have a CCA that matches the domain of P with R and a CCA that matches the range of P with T .
- (iii) If A has a DCA that matches a datatype property P in V with a relation name R of S , then A must have a CCA that matches the domain of P with R .

3.3 Transformation Rules generated by Correspondence Assertions

In this section, we introduce the notion of transformation rule and show how to interpret correspondence assertions as transformation rules. Let $\mathcal{O}=(V, \Sigma)$ be an ontology and $\mathcal{S}=(R, \Omega)$ be a relational schema, with vocabulary U . Let X be a set of *scalar variables* and T be a set of tuple variables, disjoint from each-other and from V and U .

A *literal* is a *range expression* of the form $R(t)$, where R is a relation name in U and t is a tuple variable in T , or a *built-in predicate* of one of the forms shown in Table 1. A *rule body* B is a list of literals. When necessary, we use " $B[x_1, \dots, x_k]$ " to indicate that the tuple or scalar variables x_1, \dots, x_k occur in B . We also use " $R(t), B[t, x_1, \dots, x_k]$ " to indicate that the rule body has a literal of the form $R(t)$.

A *transformation rule*, or simply a *rule*, is an expression of one of the forms:

- $C(x) \leftarrow B[x]$, where C is a class in V and $B[x]$ is a rule body
- $P(x, y) \leftarrow B[x, y]$, where P is a property and $B[x, y]$ is a rule body

Let A be a set of correspondence assertions that defines a mapping between V and S , that is, A satisfies the conditions stated in Definition 2.5. Assume that each class C in V is associated with a namespace prefix.

Table 2 shows the transformation rules *induced* by the correspondence assertions in A . For example, the rule on the right-hand side of Line 5 indicates that, for each tuple t of R such that $t.A$ is not null, one should:

- Compute the URI s of the instance of domain D of P that t represents, using the class correspondence assertion $\Psi_D: D(s) \leftarrow R(t), B_D[t, s]$, where $B_D[t, s]$ stands for " $HasURI[\Psi](t, s)$ ", if the CCA for D follows Line 1 of Table 2, or $B_D[t, s]$ stands for " $HasURI[\Psi](t, s), \sigma(t)$ ", if the CCA for D follows Line 2;
- Translate the value of A in tuple t , generating the literal v ;
- Associate v as the value for property P of s .

Table 1. Built-in predicates

Built-in predicate	Intuitive definition
$nonNull(v)$	$nonNull(v)$ holds iff value v is not null
$RDFLiteral(u, A, R, v)$	Given a value u , an attribute A of R , a relation name R , and a literal v , $RDFLiteral(u, A, R, v)$ holds iff v is the literal representation of u , given the type of A in R
$HasReferencedTuples[\varphi](t,u)$ where φ is a path from R to T	Given a tuple t of R and tuple u of T , $HasReferencedTuples[\varphi](t,u)$ holds iff u is referenced by t through path φ
$HasURI[\Psi](t,s)$ where Ψ is a CCA for a class C of \mathbf{V} , using attributes A_1, \dots, A_n of R	Given a tuple t of R , $HasURI[\Psi](t,s)$ holds iff s is the URI obtained by concatenating the namespace prefix for C and the values of $t.A_1, \dots, t.A_n$
$concat([v_1, \dots, v_n], v)$	Given a list $[v_1, \dots, v_n]$ of string values, $concat([v_1, \dots, v_n], v)$ holds iff v is the string obtained by concatenating v_1, \dots, v_n

Table 2. Transformation Rules

	Correspondence Assertion	Transformation Rule
T1	$\Psi: C \equiv R[A_1, \dots, A_n]$	$C(s) \leftarrow R(t), HasURI[\Psi](t,s)$
T2	$\Psi: C \equiv R[A_1, \dots, A_n] \sigma$	$C(s) \leftarrow R(t), HasURI[\Psi](t,s), \sigma(t)$
T3	$\Psi: P \equiv R$ where: - A has a CCA Ψ_D that matches the domain D of P with R and Ψ_D has mapping rule $D(s) \leftarrow R(t), B_D[t,s]$ - A has a CCA Ψ_N that matches the range N of P with R and Ψ_N has mapping rule $N(o) \leftarrow R(t), B_N[t,o]$	$P(s,o) \leftarrow R(t), B_D[t,s], B_N[t,o]$
T4	$\Psi: P \equiv R / \varphi$ where: - φ is a path of R to T - A has a CCA Ψ_D that matches the domain D of P with R and Ψ_D has mapping rule $D(s) \leftarrow R(t), B_D[t,s]$ - A has a CCA Ψ_N that matches the range N of P with T and Ψ_N has mapping rule $N(o) \leftarrow T(u), B_N[u,o]$	$P(s,o) \leftarrow R(t), B_D[t,s],$ $HasReferencedTuples[\varphi](t,u),$ $T(u), B_N[u,o]$
T5	$\Psi: P \equiv R / A$ where: - A has a CCA Ψ_D that matches the domain D of P with R and Ψ_D has mapping rule $D(s) \leftarrow R(t), B_D[t,s]$ - A is an attribute of R	$P(s,v) \leftarrow R(t), B_D[t,s],$ $nonNull(t.A)$ $RDFLiteral(t.A, "A", "R", v)$
T6	$\Psi: P \equiv R / \varphi / A$ where: - φ is a path of R to T - A has a CCA Ψ_D that matches the domain D of P with R and Ψ_D has mapping rule $D(s) \leftarrow R(t), B_D[t,s]$ - A is an attribute of T	$P(s,v) \leftarrow R(t), B_D[t,s],$ $HasReferencedTuples[\varphi](t,u),$ $nonNull(u.A),$ $RDFLiteral(u.A, "A", "T", v)$

T7 $\Psi: P \equiv R / \{A_1, \dots, A_m\}$ where: - A has a CCA Ψ_D that matches the domain D of P with R and Ψ_D has mapping rule $D(s) \leftarrow R(t), B_D[t, s]$ - A_1, \dots, A_m are attributes of R	$P(s, v) \leftarrow R(t), B_D[t, s],$ $nonNull(t.A_1), \dots, nonNull(t.A_m),$ $RDFLiteral(t.A_1, "A_1", "R", v_1),$ $\dots,$ $RDFLiteral(t.A_m, "A_m", "R", v_m),$ $concat([v_1, \dots, v_m], v)$
T8 $\Psi: P \equiv R / \varphi / \{A_1, \dots, A_m\}$ where: - φ is a path from R to T - A has a CCA Ψ_D that matches the domain D of P with R and Ψ_D has mapping rule $D(s) \leftarrow R(t), B_D[t, s]$ - A_1, \dots, A_m are attributes of T	$P(s, v) \leftarrow D(t), B_D[t, s],$ $HasReferencedTuples[\varphi](t, u),$ $nonNull(u.A_1), \dots, nonNull(u.A_m)$ $RDFLiteral(u.A_1, "A_1", "T", v_1),$ $\dots,$ $RDFLiteral(u.A_m, "A_m", "T", v_m),$ $concat([v_1, \dots, v_m], v)$

4 Running Example

In this section, we present the relational database schema *ISWC_REL* and the ontology *CONF_OWL*, which are used as a case study throughout the paper. We also present a set of correspondence assertions, which specifies the mapping between *CONF_OWL* and *ISWC_REL*.

Figure 1 depicts the relational schema *ISWC_REL*. Each table has a distinct primary key, whose name ends with 'ID'. *Persons* and *Papers* represent the main concepts. The attribute *conference* of *Papers* is a foreign key to *Conferences*. *Rel_Person_Paper* represents an N:M relationship between *Persons* and *Papers*. The labels of the arcs, such as *Fk_Publications*, are the names of the foreign keys.

Figure 2 depicts the ontology *CONF_OWL*, which reuses terms from four well-known vocabularies: *FOAF* (Friend of a Friend), *SKOS* (Knowledge Organization System), *VCARD* and *DC* (Dublin Core). We use the prefix "*conf*" for the new terms defined in the *CONF_OWL* ontology.

Table 3 shows a set of correspondence assertions that specifies a mapping between *CONF_OWL* and *ISWC_REL*, obtained with the help of the tool described in [19]. For example, the transformation rules induced by *CCA1*, *DCA1* and *OCA2* are (the translations from attribute values to RDF literals are omitted for simplicity):

- *CCA1* specifies that each tuple t in *Persons* produces one RDF triple:
 $\langle \text{http://example.com/person/t.perID} \rangle \text{rdf:type foaf:Person.}$
- *DCA1* specifies that each tuple t in *Persons* produces one RDF triple:
 $\langle \text{http://example.com/person/t.perID} \rangle \text{foaf:name}$
 $\text{concat}(t.firstname, t.lastname).$

- OCA2 specifies that, for each tuple t in *Person*, for each tuple t' in *Topics* such that t' is referenced by t through path [*Fk_Authors*, *Fk_Publications*, *Fk_Papers*, *Fk_Topics*], one triple of the form is generated:

$\langle \text{http://example.com/person/t.perID} \rangle \text{ conf:ResearchInterests}$
 $\langle \text{http://example.com/org/t'.topicID} \rangle.$

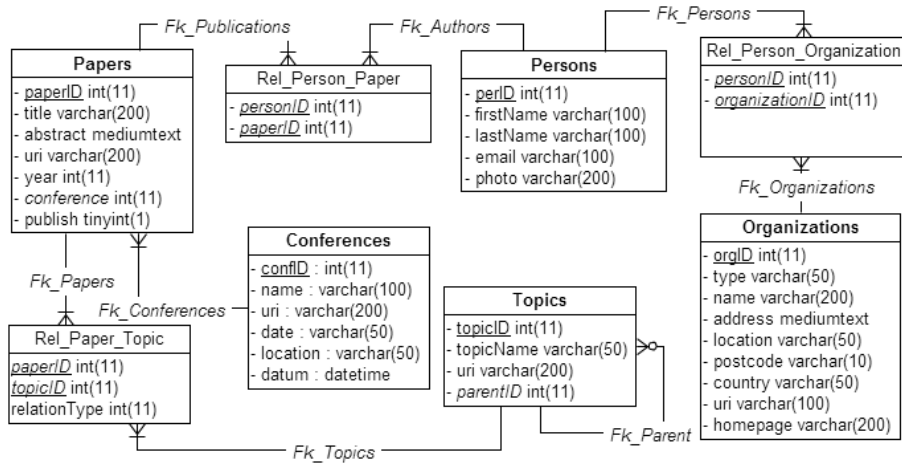


Fig. 1. ISWC_REL Database Schema.

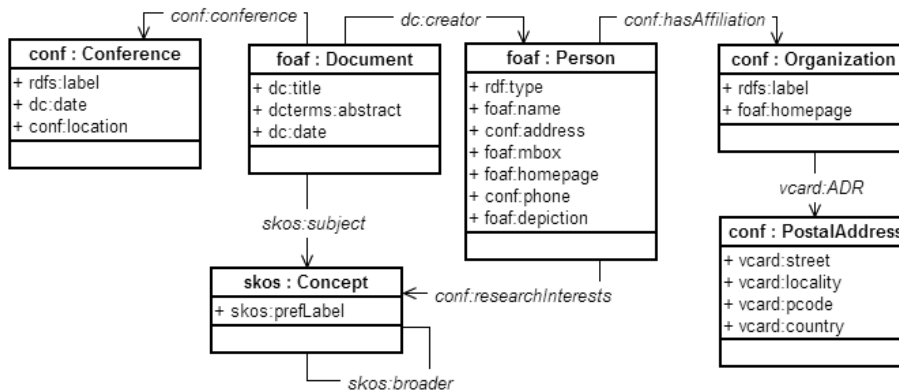


Fig. 2. CONF_OWL Target Ontology.

Table 3. Correspondence Assertions

CCA1	<i>foaf:Person</i> \equiv <i>Persons</i> [<i>perID</i>]
CCA2	<i>foaf:Document</i> \equiv <i>Papers</i> [<i>paperID</i>]
CCA3	<i>conf:PostalAddress</i> \equiv <i>Organizations</i> [<i>orgID</i>]
CCA4	<i>conf:Organization</i> \equiv <i>Organizations</i> [<i>orgID</i>]
CCA5	<i>conf:Conference</i> \equiv <i>Conferences</i> [<i>confID</i>]
CCA6	<i>skos:Concept</i> \equiv <i>Topics</i> [<i>topicID</i>]
OCA1	<i>conf:hasAffiliation</i> \equiv <i>Persons</i> / [<i>Fk_Persons</i> , <i>Fk_Organizations</i>]
OCA2	<i>conf:researchInterests</i> \equiv <i>Persons</i> / [<i>Fk_Authors</i> , <i>Fk_Publications</i> , <i>Fk_Papers</i> , <i>Fk_Topics</i>]
OCA3	<i>vcard:ADR</i> \equiv <i>Organizations</i>
OCA4	<i>skos:subject</i> \equiv <i>Papers</i> / [<i>Fk_Papers</i> , <i>Fk_Topics</i>]
OCA5	<i>conf:conference</i> \equiv <i>Papers</i> / <i>Fk_Conferences</i>
OCA6	<i>skos:broader</i> \equiv <i>Topics</i> / <i>Fk_Parent</i>
DCA1	<i>foaf:name</i> \equiv <i>Persons</i> / { <i>firstName</i> , <i>lastName</i> }
DCA2	<i>foaf:mbox</i> \equiv <i>Persons</i> / <i>email</i>
DCA3	<i>rdfs:label</i> \equiv <i>Organization</i> / <i>name</i>
DCA4	<i>foaf:homepage</i> \equiv <i>Organization</i> / <i>homepage</i>
DCA5	<i>vcard:Street</i> \equiv <i>Organizations</i> / <i>address</i>
DCA6	<i>vcard:locality</i> \equiv <i>Organizations</i> / <i>location</i>
DCA7	<i>vcard:Pcode</i> \equiv <i>Organizations</i> / <i>postcode</i>
DCA8	<i>vcard:country</i> \equiv <i>Organizations</i> / <i>country</i>
DCA9	<i>dc:title</i> \equiv <i>Papers</i> / <i>title</i>
DCA10	<i>dcterms:abstract</i> \equiv <i>Papers</i> / <i>abstract</i>
DCA11	<i>dc:date</i> \equiv <i>Papers</i> / <i>year</i>
DCA12	<i>skos:prefLabel</i> \equiv <i>Topics</i> / <i>topicName</i>
DCA13	<i>rdfs:label</i> \equiv <i>Conferences</i> / <i>name</i>

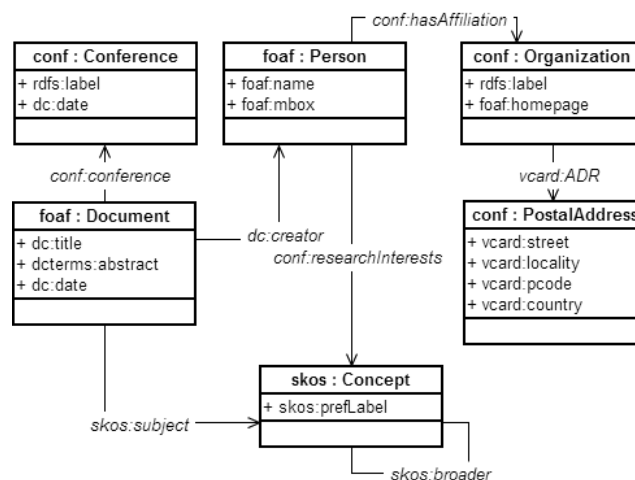


Fig. 3. ISWC_RDF Exported View Schema

5 Automatic Generation of Maintenance Rules

In this section, we present our process for generating a set of rules required for the incremental maintenance of materialized view data. The process inputs are:

- $D = (V_D, C_D)$ is the target ontology, where V_D is the vocabulary of D and C_D is the set of constraints of D
- S is the relational schema that is mapped to D
- A is a mapping, that is, a set of correspondence assertions between V_D and S

The process consists of two main steps. The first step involves the generation of the *exported RDF view schema* V , which is induced by the mapping A , as described in [5]. Figure 3 shows the exported RDF view schema $ISWC_RDF$ induced by the correspondence assertions in Table 3. The vocabulary of $ISWC_RDF$ contains all the elements of the $CONF_OWL$ ontology that match an element of $ISWC_REL$.

The second step involves the generation of a set of rules required for the incremental maintenance of the materialized view data. In following, we present how to use the view correspondence assertions to generate the set of rules required for the incremental maintenance of V .

In our approach, the process of generating the maintenance rules for V consists of the following steps:

- Obtain the set of all relations in S that are relevant to V . A relation R is relevant to V iff any correspondence assertion of V is relevant to R (see Definition 3.4).
- For each relation R that is relevant to V , three rules are generated (see Figure 4). Rule (a) is triggered by deletions on R , Rule (b) by insertions on R , and Rule (c) by updates on R . An update is treated as a deletion followed by an insertion.

When INSERT ON R Then U:= GVU_INSERT onR (r_{new}); ApplyUpdates (V, U); (a)	When DELETE ON R Then U:= GVU_DELETE onR (r_{old}); ApplyUpdates (V, U); (b)	When UPDATE ON R Then U:= GVU_DELETE onR (r_{old}); ApplyUpdates (V, U); U:= GVU_INSERT onR (r_{new}); ApplyUpdates (V, U); (c)
---	---	---

Figure 4. Rules for maintenance of View V with respect to updates on relevant relation R .

In Figure 4, the procedures $GVU_INSERT[R]$ and $GVU_DELETE[R]$ are automatically generated, at view definition time, based on the correspondence assertions of V that are relevant to R . The procedure $GVU_INSERT[R]$ takes as input a tuple r_{new} inserted in R and returns the updates necessary to maintain the view V . The procedure $GVU_DELETE[R]$ takes as input a tuple r_{old} deleted from R , and returns the updates necessary to maintain the view V . Appendix A shows the algorithm $Generate_GVU_INSERT[R]$, which takes as input the set of correspondence assertions of V that are relevant to R , and compiles the procedure “ $GVU_INSERT[R]$ ” with parameter r_{new} .

The definitions of the built-in functions used in *Generate_GVU_INSERT[R]* are defined in Table 4. The algorithm for compiling the procedure “*GVU_DELETE[R]*” is very similar, and it is omitted here for brevity.

Table 4. Built-in Functions

Built-in function	Definition
<i>GenerateRDFLiteral</i> (<i>u</i> , <i>A</i> , <i>R</i> , <i>v</i>)	$v = \text{GenerateRDFLiteral}(u, A, R)$ iff $\text{RDFLiteral}(u, A, R, v)$
<i>ObtainsReferencedTuples</i> [φ](<i>t</i>) where φ is a path from <i>R</i> to <i>T</i>	$\{ u \mid \text{HasReferencedTuples}[\varphi](t, u) \}$
<i>GenerateURI</i> [Ψ](<i>t</i>) where Ψ is a CCA for a class <i>C</i> .	$s = \text{GenerateURI}[\Psi](t)$ iff $\text{HasURI}[\Psi](t, s)$
<i>GenerateConcat</i> ([<i>v</i> ₁ , ..., <i>v</i> _{<i>n</i>}], <i>v</i>)	$v = \text{GenerateConcat}(u, A, R)$ iff $\text{concat}([v_1, \dots, v_n], v)$

Table 5 shows the procedure *GVU_INSERT[Papers]*, which returns the updates necessary to maintain the view *ISWC_RDF* with respect to insertions on relation *Papers*. By Definition 4.5, we have that CCA2, DCA9 and OCA5 are relevant to relation *Papers*. The updates required for CCA2, DCA9 and OCA5 are defined according with transformation rules T1, T4 and T5 in Table2, respectively.

Table 6 shows the procedure *GVU_INSERT[Rel_Paper_Topic]*, which returns the updates necessary to maintain the view *ISWC_RDF* with respect to insertions on relation *Rel_Paper_Topic*. By Definition 4.5, we have that OCA2 and OCA4 are relevant to relation *Rel_Paper_Topic*, because the relation *Rel_Paper_Topic* is related by the foreign key *Fk_Topics* which occurs in the path of both OCA2 and OCA4. According to case 4 of the algorithm, the updates required by OCA2 for maintaining the view *ISWC_RDF* with respect to insertions on relation *Rel_Paper_Topic* reduce to: For each tuple *t*₁ in *Persons* that is referenced by *r*_{new} through path {*Fk_Papers*, *Fk_Publications*, *Fk_Author*}, recompute all *conf:research_interests* of *t*₁ according to transformation rule T5. Note that the property *conf:research_interests* of other tuples in *Papers* is not affected by insertions on relation *Rel_Paper_Topic*.

Table 5. Procedure *GVU_INSERT[Papers]*

<pre> U := ∅; /* Updates required by CCA2 (lines 2-4 of Generate_GVU_Insert[R] algorithm) s := GenerateURI[CCA2](r_{new}); U := U ∪ { "InsertTriple(s, "rdf:type", "foaf:Document");" }; If s ≠ "" then{ /* Updates required by OCA5 (lines 11-16 of Generate_GVU_Insert[R] algorithm) Q = ObtainReferencedTuples[FK_Conferences](r_{new}); For each t in Q do { o := GenerateURI[CCA5](t); U := U ∪ { "If conf:conference(o) then InsertTriple(s, "conf:conference", o);" }; </pre>
--

```

/* Updates required by DCA9 (lines 21-24 of Generate_GVU_Insert[R] algorithm)
If nonNull(rnew.title) then {
  v := GenerateRDFLiteral(rnew.title, "title", "Papers");
  U := U ∪ {InsertTriple(s, "dc:title", v)};
};
Return(U).

```

Table 6. Procedure *GVU_Insert[Paper_Topic]*_

```

U := ∅;
/* Updates required by OCA2 (lines 38-48 of Generate_GVU_Insert[R] algorithm)
Q1 = ObtainReferencedTuple[Fk_Papers, Fk_Publications, Fk_Author](rnew);
For each t1 in Q1 do {
  s = GenerateURI[CCA1](t1);
  U := U ∪ {DeleteTriple(s, "conf:research_interests", ?o)};
  Q2 = ObtainReferencedTuple[Fk_Authors, Fk_Publications, Fk_Papers, Fk_Topics](t1);
  For each t2 in Q2 do {
    o = GenerateURI[CCA6](t2);
    U := U ∪ {If skos:concept(o) then InsertTriple(s, "conf:research_interests", o)};
  }
/* Updates required by OCA4 (lines 38-48 of Generate_GVU_Insert[R] algorithm)
Q1 = ObtainReferencedTuple[FK_Papers](rnew);
For each t1 in Q1 do {
  s = GenerateURI[CCA2](t1);
  U := U ∪ {DeleteTriple(s, "skos:Subject", ?o)};
  Q2 = ObtainReferencedTuple[Fk_Papers, Fk_Topics](t1);
  For each t2 in Q2 do {
    o = GenerateURI[CCA6](t2);
    U := U ∪ {If skos:concept(o) then InsertTriple(s, "skos:Subject", o)}}
Return(U).

```

Table 7. Procedure *GVU_Insert[Organizations]*_

```

U := ∅;
/* Updates required by CCA3 (lines 2-4 of Generate_GVU_Insert[R] algorithm)
s := GenerateURI[CCA3](rnew);
U := U ∪ {InsertTriple(s, "rdf:type", "conf:PostalAddress")};
If s ≠ "" then {
  /* Updates required by DCA5 (lines 17-20 of Generate_GVU_Insert[R] algorithm)
  If nonNull(rnew.address) then {
    v := GenerateRDFLiteral(rnew.address, "address", "Organizations");
    U := U ∪ {InsertTriple(s, "vcard:Street", v)};
  }
}

```

```

/* Updates required by DCA6 (lines 21-24 of Generate_GVU_Insert[R] algorithm)
  If nonNull(rnew.location) then {
    v := GenerateRDFLiteral(rnew.location, "location", "Organizations");
    U := U ∪ {"InsertTriple(s, "vcard:locality", v);"};
/* Updates required by DCA7 (lines 21-24 of Generate_GVU_Insert[R] algorithm)
  If nonNull(rnew.postcode) then {
    v := GenerateRDFLiteral(rnew.postcode, "postcode", "Organizations");
    U := U ∪ {"InsertTriple(s, "vcard:Pcode", v);"};
/* Updates required by DCA8 (lines 21-24 of Generate_GVU_Insert[R] algorithm)
  If nonNull(rnew.country) then {
    v := GenerateRDFLiteral(rnew.country, "country", "Organizations");
    U := U ∪ {"InsertTriple(s, "vcard:country", v);"};
/* Updates required by CCA4 (lines 2-4 of Generate_GVU_Insert[R] algorithm)
s := GenerateURI[CCA4](rnew);
U := U ∪ {"InsertTriple(s, "rdf:type", "conf:Organization");"};
If s ≠ "" then{
  /* Updates required by OCA3 (lines 21-24 of Generate_GVU_Insert[R] algorithm)
    o := GenerateURI[CCA3](rnew);
    U := U ∪ {"If conf: PostalAddress(o) InsertTriple(s, "vcard:ADR", o);"};
  /* Updates required by DCA3 (lines 21-24 of Generate_GVU_Insert[R] algorithm)
  If nonNull(rnew.title) then {
    v := GenerateRDFLiteral(rnew.name, "name", "Organizations");
    U := U ∪ {"InsertTriple(s, "rdfs:label", v);"};
  /* Updates required by DCA4 (lines 21-24 of Generate_GVU_Insert[R] algorithm)
  If nonNull(rnew.homepage) then {
    v := GenerateRDFLiteral(rnew.homepage, "homepage", "Organizations");
    U := U ∪ {"InsertTriple(s, "foaf:homepage", v);"};
Return(U).

```

6 Conclusions and Future Work

In this paper, we argued that, based on view correspondence assertions, we can automatically and efficiently identify all relations that are relevant to a view. Using view correspondence assertions, we can also define the rules required for maintaining a view with respect to updates on a relevant relation. Finally, we indicated how the rules can be automatically generated from the correspondence assertions.

We emphasize that, in our approach, the rules are responsible for directly propagating the changes to the materialized view. Our approach is effective for an externally maintained view because: the view maintenance rules are defined at view definition time; no access to the materialized view is required to compute the view maintenance

statements propagated by the rules; and the propagated view maintenance statements do not require any additional queries over the data source to maintain the view.

We are currently working on the development of a tool to automate the generation of incremental view maintenance rules.

References

1. Abiteboul, S., McHugh, J., Rys, M., Vassalos, V., Wiener, J. L., 1998. Incremental Maintenance for Materialized Views over Semistructured Data. In *VLDB*, pp. 38–49.
2. Ali, M. A., Fernandes, A. A., Paton, N. W., 2000. Incremental Maintenance for Materialized OQL Views. In *DOLAP*, pp. 41–48.
3. Bernstein, P. A. and Melnik, S., 2007. Model Management 2.0: Manipulating Richer Mappings. In *SIGMOD*, pp. 1-12.
4. Bohannon, P., Choi, B., Fan, W., 2004. Incremental evaluation of schema-directed XML publishing. In *SIGMOD*, pp. 13-18.
5. Casanova, M. A., Breitman, K. K., Furtado A. L., Vidal, V. M., Macedo, J. A., Gomes, R. V., Salas, P. E., 2011. The Role of Constraints in Linked Data. In *OTM*, pp. 781-799.
6. Ceri, S. and Widom, J., 1991. Deriving productions rules for incremental view maintenance. In *VLDB*, pp. 577–589.
7. Dimitrova, K., El-Sayed, M., Rundensteiner, E. A., 2003. Order-sensitive View Maintenance of Materialized XQuery Views. In *ER*, pp. 144–157.
8. Eisenberg, A., Melton, J., Kulkarni, K., Michels, J.E. and Zemke, F., 2004. SQL:2003 has been published. In *SIGMOD*, vol. 33, no. 1, pp. 119–126.
9. EL-Sayed, M., Wang, L., Ding, L., Rudensteiner, E., 2002. An algebraic approach for Incremental Maintenance of Materialized Xquery Views. In *WIDM*, pp. 88–91.
10. Fuxman, A., Hernandez, M. A., Ho, H., Miller, R. J., Papotti, P., Popa, L., 2006. Nested mappings: schema mapping reloaded. In *VLDB*, pp. 67–78.
11. Gupta, A. and Mumick, I.S., 2000. Materialized Views. *MIT Press*.
12. Jiang, H., HO, H., Popa, L., Han, W., 2007. Mapping-Driven XML Transformation. In *WWW*, pp. 1063–1072.
13. Kuno, H. A. and Rundensteiner, E. A., 1998. Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation. In *IEEE Transaction on Data and Knowledge Engineering*, vol. 10, no. 5, pp. 768–792.
14. Liefke, H. and Davidson, S. B., 2000. View Maintenance for Hierarchical Semi-structured Data. In *DaWaK*, pp. 114–125.
15. Miller, R. J., 2007. Retrospective on Clio: Schema Mapping and Data Exchange in Practice. In *International Workshop on Description Logics*.
16. Popa, L., Velegrakis, Y., Miller, R. J., Hernandez, M. A., Fagin, R., 2002. Translating Web Data. In *VLDB*, pp. 598–609.
17. Sawires, A., Tatemura, J., Po, O., Agrawal, D., Candan, K., 2005. Incremental Maintenance of Path-expression Views. In *SIGMOD*, pp. 443–454.
18. Staudt, M., Jarke, M., 1996. Incremental maintenance of externally materialized views. In *VLDB*, pp. 75–86.
19. Vidal, V., Casanova, Neto, L., M., Monteiro J., 2013. R2RML by Assertion: A Semi-Automatic Tool for Generating Customized R2RML Mappings. In *ESWC Demos*.
20. Vidal, V. M. P., Lemos, F. C. L., Araújo, V., Casanova M. A., 2008. A Mapping-Driven Approach for SQL/XML View Maintenance. In *ICEIS*, pp. 65-73

Appendix

Algorithm Generate “GVU_Insert[R]”

Input: The set of correspondence assertions relevant to relation R .

Output: Procedure “GVU_INSERT[R]” with parameter r_{new} .

```
1   $\tau := \{\text{"U"} := \emptyset, \};$ 
2  For each CCA  $\Psi_C$  that matches a class  $C$  with  $R$  do {
3    If  $\Psi_C : C \equiv R[A_1, \dots, A_n]$  then
4       $\tau := \tau + \{\text{"s"} := \text{GenerateURI}[\Psi_C](r_{new}); \text{U} := \text{U} \cup \{\text{"InsertTriple(s, \"rdf:type\", \"C\")\"};\};$ 
5    else if  $\Psi_C : C \equiv R[A_1, \dots, A_n]\sigma$  then
6       $\tau := \tau + \{\text{"if } \sigma(r_{new}) = \text{true then } \{\text{s} := \text{GenerateURI}[\Psi_C](r_{new});$ 
7         $\text{U} := \text{U} \cup \{\text{"InsertTriple(s, \"rdf:type\", \"C\")\"};\};$ 
8         $\text{else s} := \text{"\"};\};$ 
9     $\tau := \tau + \{\text{"if } s \neq \text{"\" then } \{\};$ 
10   For each_OCA  $\Psi_P : P \equiv R/\varphi$  where  $C$  is the domain of  $P$ ,  $\varphi$  is a path from  $R$  to  $T$ ,
11   and  $\Psi_N$  is the CCA that matches the range  $N$  of  $P$  with  $T$  do {
12      $\tau := \tau + \{\text{"Q"} := \text{ObtainReferencedTuples}[\varphi](r_{new}),$ 
13     +  $\{\text{"For each } t \text{ in } Q \text{ do } \{$ 
14        $\text{o} := \text{GenerateURI}[\Psi_N](t);$ 
15        $\text{U} := \text{U} \cup \{\text{"If } N(o) \text{ then \"InsertTriple(s, \"P\", o)\";\};\};$ 
16   For each_OCA  $\Psi_P : P \equiv R$  where  $C$  is the domain of  $P$ ,  $\Psi_N$  is the CCA that
17   matches the range  $N$  of  $P$  with  $R$  do {
18      $\tau := \tau + \{\text{"O"} := \text{GenerateURI}[\Psi_N](r_{new}),$ 
19     +  $\{\text{"U"} := \text{U} \cup \{\text{"If } N(o) \text{ then \"InsertTriple(s, \"P\", o)\";\};\};$ 
20   For each DCA  $\Psi_P : P \equiv R/A$  where  $C$  is the domain of  $P$  do {
21      $\tau := \tau + \{\text{"if } \text{nonNull}(r_{new}.A) \text{ then } \{$ 
22        $v = \text{GenerateRDFLiteral}(r_{new}.A, "A", "R");$ 
23        $\text{U} := \text{U} \cup \{\text{"InsertTriple(s, \"P\", v)\";\};\};$ 
24   For each DCA  $\Psi_P : P \equiv R/\varphi/A$  where  $C$  is the domain of  $P$ ,  $\varphi$  is a path from  $R$  to  $T$ ,
25   and  $\Psi_N$  is the CCA that matches the range  $N$  of  $P$  with  $T$  do
26      $\tau := \tau + \{\text{"Q"} := \text{ObtainReferencedTuples}[\varphi](r_{new}),$ 
27     +  $\{\text{"For each } t \text{ in } Q \text{ do } \{$ 
28        $\{\text{"if } \text{nonNull}(t.A) \text{ then } \{$ 
29          $v = \text{GenerateRDFLiteral}(t.A, "A", "T");$ 
30          $\text{U} := \text{U} \cup \{\text{"InsertTriple(s, \"P\", v)\";\};\};$ 
31     }
32   }
33 }
```

```

34 For each OCA  $\Psi_P : P \equiv R_1/\varphi$  where  $\varphi = [FK_1, \dots, FK_{n-1}]$  is a path from  $R_1$  to
35  $R_n$ , and  $R = R_j, 2 \leq j \leq n$ , do {
36   Let  $\varphi_1 = [FK_i, \dots, FK_1]$ ;
37   Let  $\Psi_D$  be the CCA that matches the domain  $D$  of  $P$  with  $R_1$ , and  $\Psi_N$  be
38   the CCA that matches the range  $N$  of  $P$  with  $R_n$ .
39    $\tau := \tau +$  "Q1 := ObtainReferencedTuples[ $\varphi_1$ ]( $r_{new}$ );"
40     + "For each  $t_1$  in Q1 do {
41        $s :=$  GenerateURI[ $\Psi_D$ ]( $t_1$ );
42        $U := U \cup$  {"DeleteTriple( $s, "P", ?o$ );"};
43        $Q2 :=$  ObtainReferencedTuples[ $\varphi$ ]( $t_1$ );
44       For each  $t_2$  in Q2 do {
45          $o :=$  GenerateURI[ $\Psi_N$ ]( $t_2$ );
46          $U := U \cup$  {"If N( $o$ ) then "InsertTriple( $s, "P", o$ );"};};
47     };
48 };
49 For each DCA  $\Psi_P : P \equiv R_1/\varphi/A$  where  $\varphi = [FK_1, \dots, FK_{n-1}]$  is a path from  $R_1$ 
50 to  $R_n$ , and  $R = R_j, 2 \leq j \leq n$ , do
51   Let  $\varphi_1 = [FK_i, \dots, FK_1]$ ;
52   Let  $\Psi_D$  be the CCA that matches the domain  $D$  of  $P$  with  $R_1$ , and  $\Psi_N$  be
53   the CCA that matches the range  $N$  of  $P$  with  $R_n$ .
54    $\tau := \tau +$  "Q1 := ObtainReferencedTuples[ $\varphi_1$ ]( $r_{new}$ );"
55     + "For each  $t_1$  in Q1 do {
56        $s :=$  GenerateURI[ $\Psi_D$ ]( $t_1$ );
57        $U := U \cup$  {"DeleteTriple( $s, "P", ?o$ );"};
58        $Q2 :=$  ObtainReferencedTuples[ $\varphi$ ]( $t_1$ );
59       For each  $t_2$  in Q2 do {
60         "if nonNull( $t_2.A$ ) then {
61            $v =$  GenerateRDFLiteral( $t_2.A, "A", "R_n$ ");
62            $U := U \cup$  {"InsertTriple( $s, "P", v$ );"};};
63     };
64 Return( $\tau$ ).

```