Computing Changesets for RDF Views of Relational Data

Vânia M. P. Vidal[✉1], Narciso Arruda[1], Matheus Cruz[1], Marco A. Casanova[2], Carlos Brito[1], Valéria M. Pequeno[3]

[1] Federal University of Ceará, Fortaleza, CE, Brazil
{vvidal, narciso, carlos}@lia.ufc.br, matheusmayron@gmail.com
[2] Department of Informatics – Pontifical Catholic University of Rio de Janeiro, RJ, Brazil
casanova@inf.puc-rio.br
INESC-ID and Universidade Autónoma de Lisboa, Lisboa, Portugal
vmp@inesc-id.pt

Abstract. The Linked Data (LD) initiative fostered the publication of a large number of relational databases in the Linked Open Data (LOD) cloud. A general and flexible way of publishing relational data in RDF format is to create an RDF view of the underlying relational data (RDB-RDF view). An RDB-RDF view can be materialized to improve query performance and data availability. However, an RDB-RDF view must be continuously maintained to reflect dynamic updates over the relational database. The contents of an RDB-RDF view, published on the LOD cloud, can also be fully or partially replicated by other LD applications to improve their efficiency. Thus, updates on an RDB-RDF view must be propagated to maintain other LD application views. This paper proposes a framework for the live synchronization of an RDB-RDF view. In the proposed framework, rules are responsible for computing and publishing the changeset required for the RDB-RDF view to stay synchronized with the relational database. The computed changesets are then used for the incremental maintenance of the RDB-RDF views as well as application views. The paper also describes a strategy for automatically generating, based on a set of mappings, the rules for computing the correct changesets for an RDB-RDF view. Finally, it suggests an architecture and describes an implementation and experiments to validate the proposed approach.

Keywords: RDF view ·View Maintenance · Linked Data · Relational database.

1 Introduction

There is a vast content of structured data available on the Web of Data as Linked Open Data (LOD). In fact, a large number of LOD datasets are RDF views defined on top of relational databases, called *RDB-RDF views*. The content of an RDB-RDF view can be materialized to improve query performance and data availability. However, to be useful, a materialized RDB-RDF view must be continuously maintained to reflect dynamic source updates.

Furthermore, Linked Data applications, that consume data RDB-RDF view, can fully or partially replicate the contents of a materialized RDB-RDF view, by creating *RDF application views* defined over the RDB-RDF view. The generation of RDF application views improves the efficiency of applications that consume data from the LOD, and increases the flexibility of sharing information. However, the generation of RDF application views raises synchronization problems, since the original datasets can be continuously updated. Thus, updates on an RDB-RDF view must be propagated to maintain the RDF application views.

A popular strategy used by large LOD datasets to maintain RDF application views is to compute and publish changesets, which indicate the difference between two states of the dataset. Applications can then download the changesets and synchronize their local replicas. For instance, DBpedia (http://wiki.dbpedia.org) and LinkedGeoData (http://linkedgeodata.org/About) publish their changesets in a public folder.

This paper proposes a framework, based on rules, that provides live synchronization of RDB-RDF views. In the proposed framework (see Fig. 1), rules are responsible for computing and publishing the changeset required for the RDB-RDF view to stay in synchronization with the relational database. The computed changesets are used by the synchronization tools for the incremental maintenance of RDB-RDF views and application views.

The use of rules is a very effective solution for computing changesets for RDB-RDF views [2, 13]. However, creating rules that correctly compute changesets for RDB-RDF views can be a complex process, which calls for tools to automate the rule generation process. Another contribution of the paper is indeed a formal framework for automatically generating, based on a set of mappings, the rules for computing correct changesets for a wide class of RDB-RDF view. Our formalism allows us to precisely justify that the rules generated by the proposed approach correctly compute the changeset.

The remainder of this paper is organized as follows. Section 2 defines transformation rules and correspondence assertions, the formalism used to specify a mapping between the relational schema and a target ontology. Section 3 introduces an abstract notation to define and materialize RDB-RDF views with the help of correspondence assertions. Section 4 describes a strategy, based on rules, for computing changesets for an RDB-RDF view. Section 5 summarizes related work. Section 6 covers an implementation and experiments. Section 7 presents the conclusions.

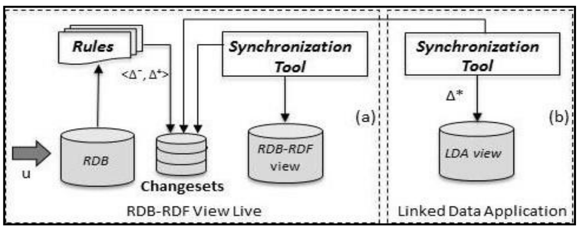

Fig. 1. Framework for Live Synchronization of RDB-RDF view.

2 Mapping Relational Data to RDF

2.1 Basic Concepts and Notation

As usual, we denote a *relation scheme* as $R[A_1,....,A_n]$ 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 $\boldsymbol{S}=(\boldsymbol{R},\boldsymbol{\Omega})$, where $\boldsymbol{R}$ is a set of relation schemes and $\boldsymbol{\Omega}$ is a set of relational constraints such that: (i) $\boldsymbol{\Omega}$ has a unique primary key for each relation scheme in $\boldsymbol{R}$; (ii) $\boldsymbol{\Omega}$ has a mandatory attribute constraint for each attribute which is part of a key or primary key; (iii) if $\boldsymbol{\Omega}$ has a foreign key of the form $F(R{:}L,S{:}K)$, then $\boldsymbol{\Omega}$ also has a constraint indicating that K is the primary key of S. The *vocabulary* of $\boldsymbol{S}$ is the set of relation names, attribute names, etc. used in $\boldsymbol{S}$. Given a relation scheme $R[A_1,....,A_n]$ 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 $\boldsymbol{S}=(\boldsymbol{R},\boldsymbol{\Omega})$ be a relational schema and R and T be relation schemes of $\boldsymbol{S}$. A list $\varphi = [F_1,...,F_{n-1}]$ of foreign key names of $\boldsymbol{S}$ is a *path from R to T* iff there is a list $R_1,...,R_n$ of relation schemes of $\boldsymbol{S}$ 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 $F_2(R_2{:}M_2,R_3{:}K_3)$.

We also recall a minimum set of concepts related to ontologies. A *vocabulary* $\boldsymbol{V}$ is a set of *classes*, *object properties* and *datatype properties*. An *ontology* is a pair $\boldsymbol{O}=(\boldsymbol{V},\Sigma)$ such that $\boldsymbol{V}$ is a vocabulary and Σ is a finite set of formulae in $\boldsymbol{V}$, the *constraints* of $\boldsymbol{O}$. 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.

2.2 Transformation Rules

In this section, we briefly present a mapping formalism, based on rules, to transform instance data from a relational database to a target ontology vocabulary. Our formalism is much simpler than mapping languages such as R2RML [6], but it suffices to capture expressive mappings. Also, the formalism incorporates concrete domains to capture concrete functions, such as "string concatenation", required for complex mappings, and concrete predicates, such as "less than", to specify restrictions. Examples of transformation rules will be presented in Table 4.

Let $\boldsymbol{O}=(\boldsymbol{V},\Sigma)$ be a target ontology and $\boldsymbol{S}=(\boldsymbol{R},\boldsymbol{\Omega})$ be a relational schema, with vocabulary $\boldsymbol{U}$. Let $\boldsymbol{X}$ be a set of *scalar variables* and $\boldsymbol{T}$ be a set of tuple variables, disjoint from each-other and from $\boldsymbol{V}$ and $\boldsymbol{U}$.

A *literal* is a *range expression* of the form $R(r)$, where R is a relation name in $\boldsymbol{U}$ and r is a tuple variable in $\boldsymbol{T}$, or a *built-in predicate*. A *rule body* B is a list of literals. When necessary, we use *"*$B[x_1,...,x_k]$*"* to indicate that the tuple or scalar variables

$x_1,\ldots,x_k$ occur in *B*. We also use *"R(r), B[r,* $x_1,\ldots,x_k$*]"* to indicate that the rule body has a literal of the form *R(r).*

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.

We assume that the rules are not mutually recursive. Hence, they act as definitions of concepts in ***V*** in terms of concepts in ***R***. Indeed, we consider a transformation rule ***r*** as a short-hand notation for the first-order sentence:

- $\forall x_1 \ldots \forall x_k$ ($C(x_1) \Leftrightarrow B[x_1 \ldots x_k]$), if ***r*** is of the form $C(x_1) \leftarrow B[x_1 \ldots x_k]$, where $x_1 \ldots x_k$ are the variables that occur in *B*
- $\forall x_1 \ldots \forall x_k$ ($P(x_1,x_2) \Leftrightarrow B[x_1 \ldots x_k]$), if ***r*** is of the form $P(x_1,x_2) \leftarrow B[x_1 \ldots x_k]$, where $x_1 \ldots x_k$ are the variables that occur in *B.*

Therefore, the mapping rules have a first-order semantics. In particular, we stress that the predicate function symbols have a fixed interpretation.

2.3 Correspondence Assertions

This section presents a more concise abstract syntax, based on correspondence assertions (CA) [14], for representing rules to transform data from a relational database to a target ontology vocabulary. The RDB-RDF 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-equijoin queries, and support most types of data restructuring that are commonly found when transforming relational data to RDF. Moreover, the CAs suffice to capture all R2RML mapping patterns proposed in the literature which we are aware of [11].

Let ***S***=(***R***,***Ω***) be a relational schema and ***O***=(***V***,*Σ*) be an ontology. Assume that *Σ* has constraints defining the domain and range of each property. Briefly, there are three types of CAs (see Table 1):

- *Class Correspondence Assertions* (CCA), which map tuples of relations to instances of classes;
- *Object Property Correspondence Assertions* (OCA), which map relationships among tuples, through a path, to instances of object properties;
- *Datatype Property Correspondence Assertions* (DCA), which map values of attributes to values of datatype properties.

Table 1. Transformation Rules.

CA	**Notation**	**Transformation Rule (TR)**
CCA	Ψ: $C \equiv R[A_1,\ldots,A_n]\ \delta$, where: - Ψ is the *name* of the CCA - *R* is a relation name of ***S***, - $A_1,\ldots,A_n$ are the attributes of the primary key of *R*, and	$C(s) \leftarrow R(r), HasURI[\Psi](r,s), \delta(r)$

	-δ is an optional selection over R	
OCA	Ψ: $P \equiv R\,/\,\varphi$, where: - Ψ is the *name* of the OCA - P is an object property of $\boldsymbol{V}$ - φ is an optional path from R to relation T	$P(s,o) \leftarrow R(r), B_D[r,s], HasReferencedTuples[\varphi](r,u), T(u), B_N[u,o]$, where $D(s) \leftarrow R(r), B_D[r,s]$ is the rule for the CCA that matches the domain D of P with R. $N(o) \leftarrow T(u), B_N[u,o]$ is the rule for the CCA that matches the range N of P with T .
DCA	Ψ: $P \equiv R\,/\,\varphi\,/\{A_1,...,A_n\}/F$, where: - Ψ is the name of the DCA - P is a datatype prop. of $\boldsymbol{V}$ - R is a relation name of $\boldsymbol{S}$ - φ is a path from R to T - $A_1,...,A_n$ are attributes of T - F is function that transforms values of attributes $A_1,...,A_n$ to values of property P - φ and F are optional	$P(s,o) \leftarrow R(r), B_D[r,s], HasReferencedTuples[\varphi](r,u), T(u), nonNull(u.A_1),...,nonNull(u.A_n), RDFLiteral(u.A_1, \text{“}A_1\text{”}, \text{“}T\text{”}, v_1), ..., RDFLiteral(u.A_n, \text{“}A_n\text{”}, \text{“}T\text{”}, v_n), F([v_1,..,v_n],v)$, *where*, $D(s) \leftarrow R(r), B_D[r,s]$ is the rule for the CCA that matches the domain D of P with R.

Table 1 shows the formalism to express each type of CA and the transformation rules *induced* by the CA. The body of a rule induced by a CA Ψ is always an expression of the form "$R(r), B[r,x_1,...,x_k]$", where R is called the *Pivot Relation* of Ψ and r the *pivot* variable. Table 2 lists the definitions of the built-in predicates adopted.

A *mapping* between $\boldsymbol{V}$ and $\boldsymbol{S}$ is a set $\boldsymbol{A}$ of correspondence assertions such that:

(i) If $\boldsymbol{A}$ has an OCA of the form $P \equiv R/\varphi$, where φ is a path from R to T, then $\boldsymbol{A}$ must have a CCA that matches the domain of P with R and a CCA that matches the range of P with T.

(ii) If $\boldsymbol{A}$ has a DCA that matches a datatype property P in $\boldsymbol{V}$ with a relation name R of $\boldsymbol{S}$, then $\boldsymbol{A}$ must have a CCA that matches the domain of P with R.

To make the paper self-contained, we introduce an abstract notation to define *RDB-RDF views* with the help of correspondence assertions.

Definition 1. An *RDB-RDF view* is a triple $\boldsymbol{V} = (O_V, \boldsymbol{S}, M_V)$, where:

- O_V is the vocabulary of the view
- $\boldsymbol{S}$ is a source relational schema
- M_V is the *mapping* between $\boldsymbol{V}$ and $\boldsymbol{S}$

Note that, by definition, since M_V is the *mapping* between $\boldsymbol{V}$ and **S**, it satisfies conditions (i) and (ii) introduced in this section. The problem of generating the correspondence assertions is addressed in [13, 14] and is out of the scope of this work. In [14], we proposed an approach to automatically generate R2RML mappings, based on a set of correspondence assertions. The approach uses relational views as a middle layer, which facilitates the R2RML generation process and improves the maintainability and consistency of the mapping.

Table 2. 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$[\varphi](r,u)$ where φ is a path from *R* to *T*	Given a tuple *r* of *R* and tuple *u* of *T*, *HasReferencedTuples*$[\varphi](r, u)$ holds iff *u* is referenced by *r* through path φ
HasURI$[\Psi](r,s)$ where Ψ is a CCA for a class *C* of ***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 $t.A_1,\dots,t.A_n$

2.4 Case Study: *MusicBrainz_RDF* View

MusicBrainz (http://musicbrainz.org/doc/About) is an open music encyclopedia that collects music metadata and makes it available to the public. The MusicBrainz Database is built on the PostgreSQL relational database engine and contains all of MusicBrainz music metadata. This data includes information about artists, release groups, releases, recordings, works, and labels, as well as the many relationships between them. Fig. 2 depicts a fragment of the MusicBrainz relational database schema (for more information about the original scheme, see https://wiki.musicbrainz.org/MusicBrainz_Database/Schema). Each table has a distinct primary key, whose name ends with 'ID'. *Artist*, *Medium, Release* and *Track* represent the main concepts. *Artist_Credit_Name* represents an N:M relationship between *Artist* and *Artist_Credit*. The labels of the arcs, such as *Fk_1,* are the names of the foreign keys.

In our case study, we will use a RDB-RDF view, called *MusicBrainz_RDF*, which is defined over the relational schema in Fig. 2. Fig. 3 depicts the ontology used for publishing the *MusicBrainz_RDF* view. It reuses terms from three well-known vocabularies, *FOAF* (Friend of a Friend), *MO* (Music Ontology) and *DC* (Dublin Core).

Table 3 shows a set of CAs that specifies a mapping between the relational schema in Fig. 2, and the ontology in Fig. 3, obtained with the help of the tool described in [14]. Table 4 shows the rules for CAs in Table 3 such that *Artist* is the pivot relation (we omit the translations from attribute values to RDF literals for simplicity). The rules for the CAs in Table 4 specifies that each tuple *r* in *Artist* produces the following RDF triple:

- *<http://musicbrainz.org/artist/r.gid> rdf:type mo:MusicArtist* . (*CCA1*), where the URI *http://musicbrainz.org/artist/t.gid is* obtained as defined in Table 1.
- *<http://musicbrainz.org/artist/r.gid> foaf:name r.name*. (*DCA1*)
- *<http://musicbrainz.org/artist/t.gid> foaf:made*
  < http://musicbrainz.org/track/r.trID,* where *r** is a tuple in *Track* such that *r** is referenced by *r* through path φ=[*fk_1, fk_2, fk_3*]. (*OCA1*).

The R2RML mappings, generated by the CAs in Table 3, are presented in [12].

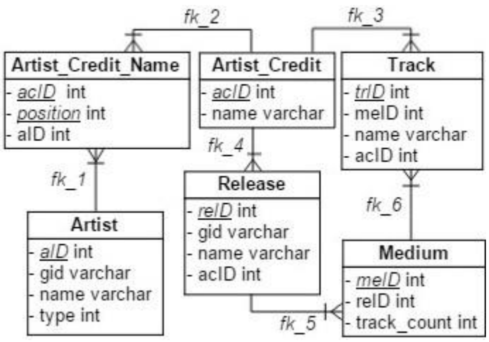

Fig. 2. Fragment of *MusicBrainz* Schema

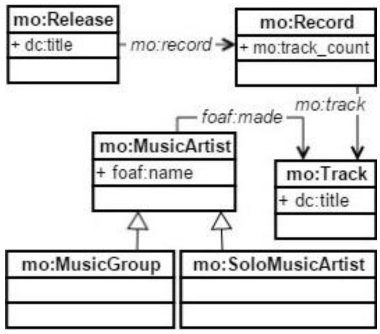

Fig. 3. *MusicBrainz_RDF* View Ontology.

Table 3. Correspondence Assertions.

CCA1	*mo:MusicArtist ≡ Artist[gid]*
CCA2	*mo:SoloMusicArtist ≡ Artist[gid][type=1]*
CCA3	*mo:MusicGroup ≡ Artist[gid][type=2]*
CCA4	*mo:Record ≡ Medium[reID]*
CCA5	*mo:Track ≡ Track[trID]*
CCA6	*mo:Release ≡ Release[gid]*
OCA1	*foaf:made ≡ Artist /[fk_1, fk_2, fk_3]*
OCA2	*mo:track ≡ Medium / [fk_6]*
OCA3	*mo:record ≡ Release / [fk_5]*
DCA1	*foaf:name ≡ Artist / name*
DCA2	*mo:track_count ≡ Medium / track_count*
DCA3	*dc:title ≡ Track / name*
DCA4	*dc:title ≡ Release / name*

Table 4. Transformation Rules for CCA1, OCA1 and DCA1.

CCA1	*mo:MusicArtist(s) ←artist(r), HasURI[CCA1](r, s)*
OCA1	*foaf:made(s,g) ←artist(r), HasURI[CCA1](r, s),* *hasReferencedTuples[fk_1, fk_2, fk_3](r, f),* *hasURI[CCA5](f, g)*
DCA1	*foaf:name(s,v) ← artist(r), HasURI[CCA1](r, s),* *nonNull(r.name),* *RDFLiteral(r.name,"name", "artist",v)*

3 Materialization of RDB-RDF View

A materialization of an RDB-RDF view requires translating source data into the RDB-RDF view vocabulary as specified by the mapping rules. In the rest of this paper, $\sigma_S(t)$ denotes the state of S in time t, where S can be a source dataset, a relation or a RDB-RDF view.

In the following definitions, let $\boldsymbol{V} = (O_V, \boldsymbol{S}, M_V)$ be an RDB-RDF view and t be a time value.

Definition 2. Let Ψ be a CA in M_V, where R is the pivot relation of Ψ and $\boldsymbol{T}$ is the transformation rule induced by Ψ. Let r be a tuple in $\sigma_{\boldsymbol{R}}(t)$, and let $\boldsymbol{T}[r]$ be the transformation rule obtained by substituting the pivot tuple variable in the body of $\boldsymbol{T}$ by tuple r. $\Psi[r](t)$ is the set of triples that are produced by rule $\boldsymbol{T}[r]$ against $\sigma_{\boldsymbol{S}}(t)$. (See [12] for a detailed formal definition).

Definition 3 (*RDF state of a tuple*). Let r be a tuple in $\sigma_{\boldsymbol{R}}(t)$, where R is a relation in $\boldsymbol{S}$. The *RDF state* of tuple r w.r.t $\boldsymbol{V}$ at time t, denoted $M_V[r](t)$, is defined as:

- $M_V[r](t) = \{x \,/\, x$ is a triple in $\Psi[r](t)\}$, if R is the pivot relation of a CA Ψ in M_V
- $M_V[r](t) = \emptyset$, if R is not the pivot relation of any assertion Ψ in M_V

Definition 4 (*Materialization of V*). The *materialization* or *state* of $\boldsymbol{V}$ at time t, denoted $M_V(t)$ is defined as:
$M_V(t) = \{x \,/\, x$ is a triple in $M_V[r](t)$, where r is a tuple in $\sigma_R(t)$ and R is the pivot relation of a CCA in $M_V\}$.

Referring to our case study, consider that the mappings defined by the CAs in Table 3. Suppose that current state of the relations in Fig. 2 is that shown in Fig. 4. Table 5 shows the materialization or state of *MUSICBRAINZ_RDF* view for the current state of relations in Fig 4.

artist

aID	gid	name	type
a1	ga1	Kungs	1
a2	ga2	Cookin' on 3 B.	2
a3	ga3	Kylie Auldist	1

artist_credit_name

acID	pos	aID
ac1	1	a1
ac1	2	a2
ac2	1	a2
ac2	2	a3

artist_credit

acID	name
ac1	Kungs vs. Cookin' on 3 B.
ac2	Cookin' on 3 B. feat. Kylie Auldist

track

trID	meID	name	acID
t1	m1	This Girl	ac2

medium

meID	reID	track_count
m1	r1	12

release

reID	gid	name	acID
r1	gr1	Layers	ac1

Fig. 4. State Example for Relations in Fig 2.

Table 5. Materialization of *MUSICBRAINZ_RDF*

$M_V(t_0)$ = {	(mbz:ga3 ***foaf:made*** mbz:t1);
(mbz:ga1 ***rdf:type*** *mo:MusicArtist*);	(mbz:ga3 ***rdf:type*** *mo:SoloMusicArtist*);
(mbz:ga1 ***foaf:name*** “Kungs”);	(mbz:m1 ***rdf:type*** *mo:Record*);
(mbz:ga1 ***rdf:type*** *mo:SoloMusicArtist*);	(mbz:m1 ***mo:track_count*** 12);
(mbz:ga2 ***rdf:type*** *mo:MusicArtist*);	(mbz:m1 ***mo:track*** mbz:t1);
(mbz:ga2 ***foaf:name*** “Cookin’s on 3 B.”);	(mbz:gr1 ***rdf:type*** *mo:Release*);
(mbz:ga2 ***foaf:made*** mbz:t1);	(mbz:gr1 ***dc:title*** “Layers”);
(mbz:ga2 ***rdf:type*** *mo:MusicGroup*);	(mbz:gr1 ***mo:record*** mbz:m1);
(mbz:ga3 ***rdf:type*** *mo:MusicArtist*);	(mbz:t1 ***rdf:type*** *mo:Track*);
(mbz:ga3 ***foaf:name*** “Kylie Auldist”);	(mbz:t1 ***dc:title*** “This Girl”)}

4 Computing Changesets for RDB-RDF Views

In this section, we first formalize the problem of computing a changeset of an RDB-RDF view when updates occur in the source database. Then, we present a strategy, based on rules, for computing changeset for an RDB-RDF view.

4.1 Correct Changesets

Let $\boldsymbol{V} = (O_V, \boldsymbol{S}, M_V)$ be an RDB-RDF view. Consider an update u_S on a relation R of $\boldsymbol{S}$, and assume that u_S is an insertion, a deletion or an update of a single tuple. Let t_0 and t_1 be the time immediately before and after the update u_S, respectively. Let $\sigma_{\boldsymbol{S}}(t_0)$ and $\sigma_{\boldsymbol{S}}(t_1)$ be the states of $\boldsymbol{S}$ respectively at t_0 and t_1, and let $M_{\boldsymbol{V}}(t_0)$ and $M_{\boldsymbol{V}}(t_1)$ be the materialization of $\boldsymbol{V}$ respectively at t_0 and t_1.

A correct changeset for $\boldsymbol{V}$ between $M_{\boldsymbol{V}}(t_0)$ and $M_{\boldsymbol{V}}(t_1)$, *denoted* $\Delta_{\boldsymbol{V}}(t_0, t_1)$, is a pair $<\Delta^{-}_{\boldsymbol{V}}(t_0, t_1), \Delta^{+}_{\boldsymbol{V}}(t_0, t_1)>$, where: $\Delta^{-}_{\boldsymbol{V}}(t_0, t_1)$ is a set of triples removed from $\boldsymbol{V}$, Δ^{+} *is* a set of triples added to $\boldsymbol{V}$, that satisfies(see the diagram in Fig. 5):

$$M_{\boldsymbol{V}}(t_1) = (M_{\boldsymbol{V}}(t_0) - \Delta^{-}_{\boldsymbol{V}}(t_0, t_1)) \cup \Delta^{+}_{\boldsymbol{V}}(t_0, t_1)$$

$$\begin{tikzcd} M_V(t_0) \arrow[r, "\Delta_V(t_0, t_1)"] & (M_V(t_0) - \Delta^{-}_V(t_0, t_1)) \cup \Delta^{+}_V(t_0, t_1) \\ & \overset{=\,?}{M_V(t_1)} \\ \sigma_S(t_0) \arrow[uu, "Mv"] \arrow[r, "u_S"] & \sigma_S(t_1) \arrow[u, "Mv"'] \end{tikzcd}$$

Fig. 5. The problem of computing RDB-RDF view changeset

4.2 A Strategy to Compute Correct Changesets

In this section, we present a strategy, based on rules, for computing correct changesets for an RDB-RDF view $\boldsymbol{V} = (O_V, \boldsymbol{S}, M_V)$.

In our strategy, we first have to identify the relations in $\boldsymbol{S}$ that are relevant for $\boldsymbol{V}$, that is, the relations whose updates might possibly affect the state of the view $\boldsymbol{V}$, as in Definition 5 and Lemma 1 below.

Definition 5: Let R and R* be a relation in $\boldsymbol{S}$. Let Ψ *be a CA* in M_V.

(i) *R is relevant to Ψ* iff R appears in the body rule of Ψ.
(ii) R is *relevant for* **V** iff R is relevant to some CA in M_V.
(iii) *R is relevant to R* w.r.t Ψ iff R is relevant to Ψ and R* is* the pivot relation of Ψ.

Lemma 1: Let u be an update on table R, and let t_0 and t_1 be the time immediately before and after u, respectively. If R is not relevant to $\boldsymbol{V}$, then $M_{\boldsymbol{V}}(t_0) = M_{\boldsymbol{V}}(t_1)$. (Proof in [12]).

Thus, based on Lemma 1, we focus our attention only on the relations that are relevant for $\boldsymbol{V}$. For each such relation R, we define triggers that are fired immediately before and immediately after an update on *R*, called *before* and *after triggers*, respectively, and which are such that:

BEFORE Trigger: Computes the set $\Delta^{-}_{\mathbf{V}}(t_0, t_1)$ using $\sigma_{\mathbf{S}}(t_0)$ (database state BEFORE the update).

AFTER Trigger: Computes the set $\Delta^{+}_{\mathbf{V}}(t_0, t_1)$ using $\sigma_{\mathbf{S}}(t_1)$ (the database state AFTER the update).

From Definition 4, to materialize a RDB-RDF view, one has to compute the RDF state of the tuples in the pivot relations. The key idea of our strategy for computing the changesets is to re-materialize only the RDF_State of the tuples in the pivot relations that might possibly be affected by the update. Thus, using $\Delta^{-}_{\mathbf{V}}(t_0, t_1)$ and $\Delta^{+}_{\mathbf{V}}(t_0, t_1)$ one should be able to compute the new RDF state of the tuples that are relevant to the update. See Definitions 6 e 7, Lemma 2 and Theorem 1 in following.

Definition 6: Let Ψ *be a CA* in M_V with path φ. Let *R* and *R** be relations in ***S***, where *R is relevant to R* w.r.t* Ψ (Def. 5(iii)). A tuple *r* in $\sigma_{\boldsymbol{R}}(t)$ *is relevant to a tuple r* in* $\sigma_{\boldsymbol{R*}}(t)$ *w.r.t* Ψ iff *r** references *r* through a not empty path φ*, where φ* is a prefix of φ.

For Definition 7 and Lemma 2, let *u* be an update on table *R*, and let t_0 and t_1 be the time immediately before and after *u*, respectively.

Definition 7: Let r_{new} and r_{old} be the new and old state of the updated tuple in R.

(i) $\boldsymbol{A_{NEW}}$= {*r* / r** is a tuple in $\sigma_{\boldsymbol{R*}}(t_0)$, where *R is relevant to R* w.r.t* a CA Ψ *in* M_V and ($r^*=r_{new}$ or (r_{new} is relevant to *r* w.r.t* Ψ (Def. 6))}.

(ii) $\boldsymbol{A_{OLD}}$ = {*r* / r** is a tuple in $\sigma_{\boldsymbol{R*}}(t_0)$, where *R is relevant to R* w.r.t* a CA Ψ *in* M_V and ($r^*=r_{old}$ or (r_{old} is relevant to *r* w.r.t* Ψ (Def. 6))}.

(iii) Let *r** be a tuple in $\sigma_{\boldsymbol{R*}}(t_0)\cup\sigma_{\boldsymbol{R*}}(t_1)$. Then, *r** is relevant to ***V*** w.r.t the update *u* iff *r** is in $\boldsymbol{A_{NEW}} \cup \boldsymbol{A_{OLD}}$.

Lemma 2: Let *r** be a tuple in $\sigma_{\boldsymbol{R*}}(t_0)\cup\sigma_{\boldsymbol{R*}}(t_1)$. If *r** is not relevant to ***V*** w.r.t the update *u* (c.f Def. 8), then $M_V[r^*](t_0)=M_V[r^*](t_1)$. (Proof in [12]).

Theorem 1: Let:

- *u* be an update on *R*
- t_0 and t_1 be the time immediately before and after *u*
- r_{new} and r_{old} be the new and old state of the updated tuple
- $\boldsymbol{A_{NEW}}$ and $\boldsymbol{A_{OLD}}$ as in Definition 7(i) and 7(ii), respectively
- $\Delta^{-}_{\mathbf{V}}(t_0, t_1)$ = {*x / x* is in $M_V[r](t_0)$ where *r* is a tuple in $\boldsymbol{A_{NEW}} \cup \boldsymbol{A_{OLD}}$ }
- $\Delta^{+}_{\mathbf{V}}(t_0, t_1)$ = {*x / x* is in $M_V[r](t_1)$ where *r* is a tuple in $\boldsymbol{A_{NEW}} \cup \boldsymbol{A_{OLD}}$ }

Then, $M_V(t_1) = (M_V(t_0) - \Delta^{-}_{\mathbf{V}}(t_0, t_1)) \cup \Delta^{+}_{\mathbf{V}}(t_0, t_1)$. (See proof in [12])

According to Lemma 2, an update may affect the RDF_State of a tuple in a pivot relation iff the tuple is relevant to ***V*** w.r.t the update (Def. 7(iii)). Fig. 6 shows the templates of the triggers associated with an update on a relation *R*. For example, consider *u*, the UPDATE on R were r_{old} and r_{new} are the old and new state of the updated tuple, respectively. Before the update, Trigger (a) is fired, and the Procedure *COMPUTE_* Δ^{-}*[R]* computes $\Delta^{-}_{\mathbf{V}}(t_0, t_1)$ which contains the OLD RDF_State of the tuples that are relevant to ***V*** w.r.t update *u* (see Theorem 1). After the update, Trigger (b) is fired. Using the database state after the update, the Procedure *COMPUTE_* Δ^{+}[*R*] computes $\Delta^{+}_{V}(t_0, t_1)$ which contains the new RDF_State of the tuples that are relevant to ***V*** w.r.t update *u* (see Theorem 1). It is important to notice, that the procedures *COMPUTE_* Δ^{-}[*R*] and *COMPUTE_* Δ^{+} [*R*] are automatically generated, at view definition time, based on the correspondence assertions of *V*. In [12] we show the algorithms which takes as input the set of correspondence assertions of *V* and compiles the procedures *“COMPUTE_* Δ^{-}[*R*]*”* and “*COMPUTE_* Δ^{+} [*R*]*”*.

Theorem 1 shows that the triggers in Fig. 6 correctly compute the changesets for an update on a relation *R*. Triggers for insertions and deletions are similarly defined and are omitted here. A trigger for an insertion computes only $\boldsymbol{A_{NEW}}$, while a trigger for a deletion computes only $\boldsymbol{A_{OLD}}$ (see Def. 7). Note that our strategy can be easily extended to more complex type of correspondence assertions, provided the sets $\boldsymbol{A_{OLD}}$ and $\boldsymbol{A_{NEW}}$ are properly defined for more complex types of CA.

BEFORE {UPDATE } 0N R **THEN** Δ^{-}:= *COMPUTE_* Δ^{-}[*R*] (r_{old}, r_{new}); ADD Δ^{-} to changeset file of *V*. (a)	**AFTER** {UPDATE} 0N R **THEN** Δ^{+} := *COMPUTE_* Δ^{+} [*R*] (r_{old}, r_{new}); ADD Δ^{+} to changeser file of *V*. (b)

Fig. 6. Triggers to compute changeset of V w.r.t. updates on *R*

4.3 Computing Changeset Example

To illustrate this strategy, consider that state of database in Fig. 4. Also consider *u*, the UPDATE on table *Track* were r_{old} =<t_1, m_1, “This Girl”, ac_2>, and r_{new} = <t_1, m_1, “This Girl (feat. Cookin' On 3 B.)”, ac_1>. The changeset for the view *MusicBrainz_RDF* is computed in two phases as follow:

Phase 1: Executed by the BEFORE trigger (see Fig. 6(a)).

From Def. 6 e ACs in Table 3, we have:
- From OCA1: r_{old} is relevant to tuples a_2 and a_3 in table *Artist* and r_{new} is relevant to tuples a_1 and a_2 in table *Artist.*
- From *OCA2:* r_{old} and r_{new} are relevant to tuple m_1 in Table *Medium*.

From Def. 7, we have:

$\boldsymbol{A_{OLD}} = \{r_{old}, a_2, a_3, m_1>\}$.
$\boldsymbol{A_{NEW}} = \{r_{new}, a_1, a_2, m_1>\}$.

From Theorem 1, we have:

Δ^{-} ={ (mbz:t1 *rdf:type mo:Track*);(mbz:t1 dc*:title* “This Girl”);
(mbz:ga2 *rdf:type mo:MusicArtist*); (mbz:ga2 *foaf:name* “Cookin’s on 3 B.”);
(mbz:ga2 *foaf:made* mbz:t1); (mbz:ga2 *rdf:type mo:MusicGroup*);
(mbz:ga3 *rdf:type mo:MusicArtist*); (mbz:ga3 *foaf:name* “Kylie Auldist”);
(mbz:ga3 *foaf:made* mbz:t1); (mbz:ga3 *rdf:type mo:SoloMusicArtist*);
(mbz:m1 *rdf:type mo:Record*);
(mbz:m1 *mo:track_count* 12); (mbz:m1 *mo:track* mbz:t1);

(mbz:ga1 *rdf:type mo:MusicArtist*); (mbz:ga1 *foaf:name* “Kungs”);
(mbz:ga1 *rdf:type mo:SoloMusicArtist*)}

Phase 2: Executed by the AFTER trigger (Fig. 6 (b)).
From Theorem 1, we have:
Δ^+ ={ (mbz:ga2 *rdf:type mo:MusicArtist*); (mbz:ga2 *foaf:name* “Cookin’s on 3 B.”);
(mbz:ga2 foaf:made mbz:t1); (mbz:ga2 rdf:type mo:MusicGroup);
(mbz:ga3 rdf:type mo:MusicArtist); (mbz:ga3 foaf:name “Kylie Auldist”);
(mbz:ga3 rdf:type mo:SoloMusicArtist);
(mbz:t1 rdf:type mo:Track);
(mbz:t1 dc:title “ThisGirl (feat. Cookin' On 3 B.)”);
(mbz:m1 rdf:type mo:Record);
(mbz:m1 mo:track_count 12); (mbz:m1 mo:track mbz:t1);
(mbz:ga1 rdf:type mo:MusicArtist); (mbz:ga1 foaf:name “Kungs”);
(mbz:ga1 foaf:made mbz:t1); (mbz:ga1 rdf:type mo:SoloMusicArtist)}

Given V_{OLD}, the old State of the *MusicBrainz_RDF* view in Table 5, in order to stay synchronized with the new state of database, the new state of the view can computed by: $V_{NEW} = (V_{OLD} - \Delta^-) \cup \Delta^+$.

5 Related Work

The Incremental View Maintenance problem has been extensively studied in the literature for relational views [2], object-oriented views [8], semi-structured views [1], and XLM Views [3]. Despite their important contributions, none of these techniques can be directly applied to compute changesets for RDB-RDF views, since the complexity of the RDB-RDF mapping pose new challenges, when compared with other types of Views.

Comparatively less work addresses the problem of incremental maintenance of RDB-RDF views. Vidal et al. [13] proposed an incremental maintenance strategy, based on rules, for RDF views defined on top of relational data. Their approach was further modified and enhanced to compute changesets for RDB-RDF views. Although the approach in this paper uses similar formalism for specifying the view’s mappings, our strategy to compute changeset proposed in Section 4 differ considerably.

Endris at al [4] introduces an approach for interest-based RDF update propagation that can consistently maintain a full or partial replication of large LOD datasets. Faisal at al [5] presented an approach to deal with co-evolution which refers to mutual propagation of the changes between a replica and its origin dataset. Both approaches relies on the *assumption* that either the source dataset provides a tool to compute a changeset at real-time or third party tools can be used for this purpose. Therefore, the contribution of this paper is complementary and rather relevant to satisfy their assumption.

The works in [9, 10, 15] address the problem of change detection between versions of LOD datasets. In [15], a low-level change detection approach is used to report simple addition / exclude operations. In [9, 10], a high-level change detection approach is used to provide more readable deltas by humans. Despite their important contributions for understanding and analyzing the dynamicity of Web datasets, these techniques cannot be applied to compute changesets for RDB-RDF views.

Konstantinou et al [7] investigates the problem of incremental generation and storage of the RDF graph that is the result of exporting relational database contents. Their approach uses the mappings definition to identify the affected tables. Then, the whole triples map definition will be executed for all tuples in the affected table.

To the best of our knowledge, the computation of changeset for RDB-RDF views has not yet been addressed in any framework.

6 Implementation and Experiments

To test our strategy, we implemented the *LinkedBrainz Live tool* (*LBL tool*), which propagates the upadates over the *MusicBrainz* database (*MBD* database) to the LinkedMusicBrainz view (*LMB View*). The *LMB* View is intended to help MusicBrainz (http://musicbrainz.org/doc/About) to publish its database as Linked Data. The main components of the *LBL* tool (See Fig. 1) are:

- **Local *MBD* database**: We installed a local replica of the *MBD* database available on March 22, 2017. The decompressed database dump has about 7 GB and was stored in PostgreSQL version 9.4.
- ***LMB View* and Mappings**: We created the R2RML mapping for translating *MBD* data into the Music Ontology vocabulary (http://musicontology.com/), which is used for publishing the *LMB* view. We also provided SPARQL endpoint for querying *LMB* View.
- ***Triggers***: We created the triggers to implement the rules required to compute and publish the changesets, as discussed in Section 4.
- ***LBL update extractor***: This component extracts updates from the replication file provided by MusicBrainz, every hour, which contains a sequential list of the update instructions processed by the MusicBrainz database. When there is a new replication file, the updates should be extracted and then executed against the local of the *MBD* database.
- ***LBL Syncronization tool***: This component enables the *LMB View* to stay synchronized with the *MBD* database. It simply downloads the changeset files sequentially, creates the appropriate INSERT/DELETE statement and executes it against the *LMB View* triplestore.

In our preliminary experiments, we measure the time to compute the changeset for replication file with sequential number 103114, which has 4,557 updates. The total time to compute the changeset for this replication file was 5 minutes. Table 6 and Table 7 summarize our experimental results, considering each relevant relation (RR) separately. Due to space limitation we show results only for updates over RR of case study in Section 2.4 (*Artist, Medium, Release, Track, and Artist_Credit_Name*).

- Table 6 shows: The total number of tuples in the RR, the total time (in milliseconds) spent to triplify the RR, and the total number of updates on the RR.
- Table 7 shows: The average number of tuples relevant to updates on the RR, and the average time (in milliseconds) to compute the changeset <Δ^-, Δ^+> for the RR.

The experiments demonstrated that the runtime for computing the changeset is negligible, since the number of tuples that are relevant to an update is relatively small. We also analyzed the MusicBrainz replication file from one day and concluded that, in the experiments, just a small percentage of the tuples are relevant for the updates. Thus, we can conclude that, in the situation where the RDB-RDF view should be frequently updated, the incremental strategy far outperforms full re-materialization, and also the re-materialization of the affected tables [7].

Table 6. Relevant Relation of our Case Study (Fig. 2)

Relevant Relation (RR)	**Number of Tuple (k)**	**Triplification Time** (ms)	**Number of Updates**
Artist	1,189	340,721	31
Medium	1,988	290,689	398
Release	1,768	63,014	120
Track	22,087	435,693	397
Artist_Credit_Name	1,974	* not a pivot relation	14

Table 7. Changeset Computation Performance for Relevant Relations

Relevant Relation (RR)	**Avg Number of Relevant tuples**	Δ^- (avg time)(ms)		Δ^+ (avg time)(ms)	
		i	***u***	***i***	***u***
Artist	1.55	25	102	18	5
Medium	4.05	128	32	20	4
Release	1.52	63	47	6	4
Track	4.05	28	39	4	3
Artist_Credit_Name	1	49	--	5	--

7 Conclusions

In this paper, we presented a framework for providing live synchronization of an RDF view defined on top of relational database. In the proposed framework, rules are responsible for computing and publishing the changeset required for the RDB-RDF view to stay synchronized with the relational database. The computed changesets are used for the incremental maintenance of the RDB-RDF views as well as application views.

We also presented a method for automatically generating, based on a set of correspondence assertions, the rules for computing the changesets for a wide class of RDB-RDF view. We showed that, based on view correspondence assertions, we can automatic and efficiently identify all relation that are relevant to a view. Using view correspondence assertions, we can also define how to identify the tuples that are relevant to a given source update. Therefore, only the RDF_State of the relevant tuple are re-materialized. We provided a thorough formalization of our approach and indicated that the rules generated by the proposed approach correctly compute the changeset.

We also implemented the *LinkedBrainz Live tool* to validate the proposed framework. We are currently working on the development of a tool to automate the generation of the rules for computing the changesets.

Acknowledgments. This work was partly funded by CNPq under grants 303332/2013-1, 442338/2014-7 and 557128/2009-9 and by FAPERJ under grants E-26-170028-2008 and E-26/201.337/2014.

8 References

1. Abiteboul, S., McHugh, J., Rys, M., Vassalos, V., Wiener, J. L.: Incremental Maintenance for Materialized Views over Semistructured Data. In VLDB 1998, pp. 38–49 (1998)
2. Ceri, S. and Widom, J.: Deriving productions rules for incremental view maintenance. In VLDB 1991, pp. 577–589 (1991)
3. Dimitrova, K., El-Sayed, M., Rundensteiner, E.A.: Order-sensitive View Maintenance of Materialized XQuery Views. In ER 2003, pp. 144–157 (2003)
4. Endris, K.M., Faisal, S., Orlandi, F., Auer, S., Scerri, S.: Interest-based RDF update propagation. In: 14th International Semantic Web Conference - ISWC2015 (2015)
5. Faisal, S., Endris, K.M., Shekarpour, S., Auer, S.: Co-evolution of RDF Datasets. In: 16th International Conference – ICWE 2016 (2016)
6. Hert, M., Reif, G., Gall, H.: A comparison of RDB-to-RDF mapping languages. In I-Semantics'11, ACM (2011)
7. Konstantinou, N., Spanos, D.E., Kouis, D., Mitrou, N.: An approach for the incremental export of relational databases into rdf graphs. International Journal on Artificial Intelligence Tools 24(2), 1540,013 (2015)
8. Kuno, H. A. and Rundensteiner, E. A.: Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation. In IEEE TDKE, vol. 10, no. 5, pp. 768–792 (1998)
9. Papavasileiou, V., Flouris, G., Fundulaki, I., Kotzinos, D. and Christophides, V.: High-Level Change Detection in RDF(S) KBs. ACM Trans. Database Syst., 38(1), (2013)
10. Roussakis, Y., Chrysakis, I., Stefanidis, K., Flouris, G. and Stavrakas, Y.: A Flexible Framework for Understanding the Dynamics of Evolving RDF datasets. In ISWC, 2015
11. Sequeda, J., Priyatna, F., Villazón-Terrazas, B.: Relational Database to RDF Mapping Patterns. In: Workshop on Ontology Patterns (WOP2012), Boston, USA, 2012
12. Vidal, V.M.P., Arruda, N., Casanova, M.A., Brito, C., Pequeno, V.M.: Computing Changesets for RDF Views of Relational Data. Technical Report, Federal University of Ceara (2016). Available at http://tiny.cc/TechnicalReportUFC2017
13. Vidal, V.M.P., Casanova, M.A., Cardoso, D.S.: Incremental Maintenance of RDF Views of Relational Data. In ODBASE 2013, pp 572-587 (2013)
14. Vidal, V.M.P., Casanova, M.A., Eufrasio, L., Monteiro, J.M.: A semi-automatic approach for generating customized R2RML mappings. In SAC'14. ACM, pp.316-322. (2014)
15. Zeginis, D., Tzitzikas, Y. and Christophides, V.: On Computing Deltas of RDF/S knowledge bases. ACM Trans. Web, 5(3):14:1–14:36, (2011)