Incremental Maintenance of Materialized SPARQL-based Linkset Views

Elisa S. Menendez[1], Marco A. Casanova[1], Vânia M. P. Vidal[2], Bernardo Pereira Nunes[1,3], Giseli Rabello Lopes[4], and Luiz A.P. Paes Leme[5]

[1]Department of Informatics, PUC-Rio, Rio de Janeiro, RJ – Brazil
{emenendez,casanova,bnunes}@inf.puc-rio.br
[2]Computer Science Department, UFC, Fortaleza, CE – Brazil
vvidal@lia.ufc.br
[3]Department of Applied Informatics, UNIRIO, Rio de Janeiro, Brasil
bernardo.nunes@uniriotec.br
[4]Departamento de Ciência da Computação, UFRJ, Rio de Janeiro, RJ – Brasil
giseli@dcc.ufrj.br
[5]Computer Science Institute, UFF, Niteroi, RJ – Brazil
lapaesleme@ic.uff.br

Abstract. In the Linked Data field, data publishers frequently materialize *linksets* between two different datasets using link discovery tools. To create a linkset, such tools typically execute *linkage rules* that retrieve data from the underlying datasets and apply matching predicates to create the links, in an often complex process. Also, such tools do not support linkset maintenance, when the datasets are updated. A simple, but costly strategy to maintain linksets up-to-date would be to fully re-materialize them from time to time. This paper presents an alternative strategy, called *incremental*, for maintaining linksets, based on idea that one should re-compute only the links that involve the updated resources. The paper discusses in detail the incremental strategy, outlines an implementation and describes an experiment to compare the performance of the incremental strategy with the full re-materialization of linksets.

Keywords: RDF Views, Linksets, SPARQL Update, Linked Data.

1 Introduction

The Linked Data initiative defines best practices for publishing and interlinking data on the Web using RDF triples to represent the data (Berners-Lee, 2006). Briefly, a dataset is simply a set of RDF triples. A link is an RDF triple of the form (*s*,*p*,*o*), where *s* and *o* are resources defined in two distinct datasets. A linkset is a set of links. *SPARQL* is the standard query language used to query RDF datasets. A *SPARQL-based view* is a view defined by a SPARQL query.

Link discovery tools help create and materialize linksets by matching resources retrieved from two datasets. The first step to configure a link discovery tool typically defines what amounts to SPARQL-based views that specify sets of resources with use-

ful properties. We refer to such views as *catalogue views* since they act as a catalogue of resources. The second step defines a set of *linkage rules* that specify conditions that resources must fulfill to be matched.

When a dataset is updated, the maintenance of a linkset requires attention since the resources may no longer meet the graph template used in the corresponding catalogue view. A trivial case is when a resource used in a link is removed from the original dataset; in this case, the link becomes invalid and must also be removed. The work in (Casanova *et al*., 2014) specified an incremental strategy to keep linksets updated, similar to the incremental strategies for relational view maintenance.

This paper extends the work reported in (Casanova *et al*., 2014) in three directions: (1) it presents in detail an incremental strategy to maintain materialized linksets; (2) it outlines an implementation of the proposed incremental strategy; (3) it describes experiments to measure the performance of the proposed incremental strategy and compare it with a re-materialization strategy.

The paper is organized as follows. Section 2 reviews related work. Section 3 contains basic definitions and a simple example. Section 4 details the incremental strategy to maintain linksets. Section 5 outlines a tool that implements the proposed incremental strategy and describes experiments conducted to assess the effectiveness of the incremental strategy. Finally, section 6 contains the conclusions and discusses directions for future research.

2 Related Work

Several tools were developed to help solve the problem of finding links between different datasets. The LInk Discovery Framework for MEtric Spaces (LIMES) proposes algorithms that work efficiently with large knowledge bases (Ngomo and Auer, 2011). The LIMES developers started with the idea of filtering obvious non-match instances to reduce the number of comparisons and improve matching time. The Silk Linking Framework (Volz *et al*., 2009b) offers a second example.

The *Web of Data – Link Maintenance Protocol* (WOD-LMP) (Volz, *et al*., 2009a) is a protocol that helps link maintenance. It covers three use cases: (1) Link Transfer to Target – the source sends notifications to the target when a link is created or deleted; (2) Request of Target Change List – the source requests to the target a list of changes in a specified time range; (3) Subscription of Target Changes – the source sends the links notifications and the target stores this information to further notify the source about changes in the pointed resources.

DSNotify (Popitsch and Haslhofer, 2011) is a general-purpose change detection framework that notifies linked datasets about events (create, remove, move, update) in their remote resources. To deal with these changes, DSNotify uses its own OWL Lite vocabulary, called DSNotify Eventset Vocabulary, which allows a detailed description (what, how, when and why) about the events.

We note that LIMES and Silk, although popular link discovery tools, do not support linkset maintenance, whereas WOD-LMP and DSNotify deal with change notification, but not with the actual linkset maintenance, as addressed in this paper.

The work reported in this paper is also related to strategies for materialized view maintenance. In the context of relational databases, a strategy for view maintenance is called *incremental* if only part of the view is modified to reflect the updates in the database (Gupta *et al*., 1993; Staud and Jarke, 1996).

This strategy was adapted to maintain RDF views over relational databases (Vidal *et al*., 2013). In all such contexts, incremental view maintenance generally outperforms full view re-computation. However, we cannot directly adopt the familiar strategies proposed for incremental maintenance over relational datasets, since complex SPARQL updates pose new challenges, when compared with SQL updates.

The work reported in this paper is also closely related to strategies designed to maintain RDF mirrors[1], slices (Ibáñez *et al*., 2014) and views (Hung *et al*., 2004; Vidal *et al*., 2015; Endris *et al*., 2015) over RDF datasets, since the main part of our strategy is to compute the resources that affect the catalogue views used in the linksets. However, there is no work in the literature that deals with *complex* SPARQL-based views. Also, the proprietary systems that support the incremental maintenance of views, such as Oracle RDF Store, can only deal with small inserts.

Furthermore, we cannot consider that a linkset is a regular RDF view computed from two datasets, since they are materialized using complex linkage rules, which typically involve similarity measures that cannot be expressed with a SPARQL query. Hence, even if there was a solution for the maintenance of SPARQL-based views in the literature, we still would not be able to direct use it.

As already mentioned in the introduction, the work reported in this paper differs from previous work by the authors (Casanova *et al*., 2014) in three aspects. First, it presents in detail the incremental strategy to keep linksets updated, which includes a normalization process for views defined by SPARQL queries and a discussion on how to synthesize queries that compute sets of affected resources. Second, it briefly outlines an implementation of the proposed strategy. Lastly, based on the implementation, it describes experiments to measure the performance of the incremental strategy when compared with a full re-materialization strategy, a question neglected in the literature.

3 Catalogue Views and Linkset views

3.1 Basic Definitions and Notation

To make the paper self-contained, we introduce an abstract notation to define *catalogue views* and *linkset views*, based on a minimum set of simple SPARQL 1.1 constructs (Harris and Seaborne, 2013). The abstract notation is convenient since it highlights the aspects involved in the construction of materialized views and linksets.

Catalogue views and linkset views depend on the notion of a *simple construct query*, which intuitively defines the catalogue of resources. A SPARQL query F is a *simple construct query*, or a *simple query*, iff

[1] https://github.com/dbpedia/dbpedia-live-mirror

- The CONSTRUCT clause of F has exactly one template of the form "?x *rdf:type* C" and a list of templates of the form "?x P_k ?p_k", where C is a class and P_k is a property, for k=1,...,n. We say that $V_F=\{C,P_1,...,P_n\}$ is the *vocabulary* of F;
- F contains a single FROM clause, specifying the dataset used to evaluate F;
- The WHERE clause of F contains the pattern of the values that will be mapped to the resources and properties of the CONSTRUCT clause; the WHERE clause is such that it does not contain negations or the MINUS operator (Section 4.2 will discuss the reasons for restricting the WHERE clause).

A *catalogue view definition* is a pair $\boldsymbol{v}=(V,F)$, where F is a simple construct query, called the *view mapping*, and V is the vocabulary of F, called the *view vocabulary*. Whenever possible, we will simply refer to F as the view definition.

Assume that a dataset contains a single set of RDF triples. Let T be the dataset specified in the FROM clause of F and $\sigma_T(t)$ be the state of T at time t. When evaluated against $\sigma_T(t)$, the simple query F returns a set of triples, which we denote $F[\sigma_T(t)]$.

A *materialization* of $\boldsymbol{F}$ at time t is the process of computing $F[\sigma_T(t)]$ and storing it as part of a dataset. We could naturally expand the abstract notation for a simple view to indicate the dataset and provide a name for the materialization of the view.

A *linkset view definition* is a quintuple $\boldsymbol{l} = (p,F,G,\pi,\mu)$, where

- p is the link property
- F and G are simple queries whose vocabularies have the same cardinality n and whose FROM clauses specify the datasets over which $\boldsymbol{l}$ is evaluated
- π is a permutation of (1,...,n), called the *alignment* of $\boldsymbol{l}$
- μ is a $2n$-relation, called the *match predicate* of $\boldsymbol{l}$

Let $V_F=\{C,P_1,...,P_n\}$ and $V_G=\{D,Q_1,...,Q_n\}$ be the vocabularies of F and G, respectively. Intuitively, π indicates that, for each $k=1,...,n$, the match predicate will compare values of P_k with values of Q_m, where $m = \pi(k)$. The notion of alignment could be generalized to permit more sophisticated alignments and mappings.

Let T be the dataset specified in the FROM clause of F and U be the dataset specified in the FROM clause of G. We say that $\boldsymbol{l}$ is *evaluated over T and U* and that $\boldsymbol{l}$ is *from T to U*. Let $\sigma_T(t)$ and $\sigma_U(t)$ be the states of T and U at time t. The linkset view definition $\boldsymbol{l}$ induces a set of triples, denoted $\boldsymbol{l}[\sigma_T(t),\sigma_U(t)]$, as follows:

$(s,p,o) \in \boldsymbol{l}[\sigma_T(t),\sigma_U(t)]$ iff there are triples
$(s, rdf{:}type, C), (s, P_1, s_1), \ldots, (s, P_n, s_n) \in F[T]$ and
$(o, rdf{:}type, D), (o, Q_1, o_1), \ldots,(o, Q_n, o_n) \in G[U]$ such that
$(s_1, \ldots, s_n, o_{m1}, \ldots, o_{mn}) \in \mu$, where $\pi(k)$, for each k=1,...,n

Again, a *materialization* of $\boldsymbol{l}$ is the process of computing the set $\boldsymbol{l}[\sigma_T(t),\sigma_U(t)]$ and storing it as part of a dataset. Also, we could expand the abstract notation to indicate the dataset and provide a name for the materialization of a linkset view definition.

3.2 Running Example

To illustrate catalogue views and linkset views, consider the dataset called Internet Movie Database (IMDb), which contains triples about movies, actors, etc. Suppose

that IMDb has a fictitious endpoint <http://imdb.org/sparql>, with default graph <http://imdb.org/data>, and uses the ontology in Fig. 1. Also consider the DBpedia dataset, which contains triples extracted from Wikipedia pages. It uses the endpoint <http://dbpedia.org/sparql> with default graph <http://dbpedia.org> and the ontology partially presented in Fig. 1.

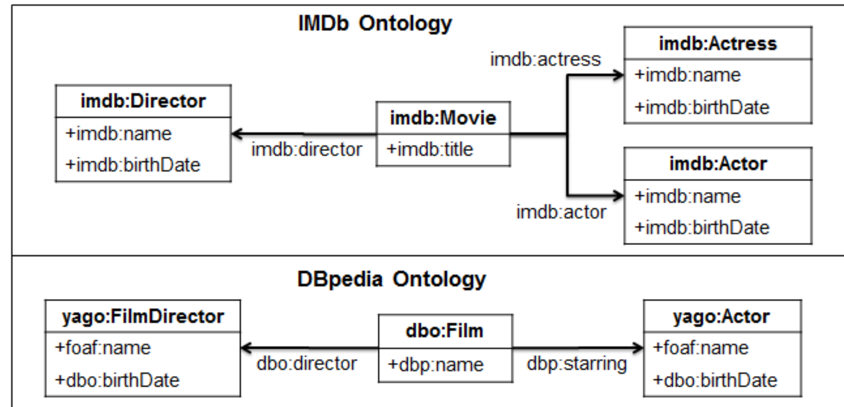

Fig. 1. Simplified fragments of IMDb and DBpedia Ontologies.

Suppose that a user wants to link the directors in *IMDb* with those in DBpedia by comparing their names and birth dates. For that purpose, s/he uses two catalogue views, *M* and *D*, respectively over IMDb and DBpedia. Suppose that *M* is:

```
CONSTRUCT { ?x rdf:type yago:FilmDirector.
            ?x foaf:name ?nm . ?x dbo:birthDate ?bt }
FROM <http://imdb.org/data>
WHERE { ?x rdf:type imdb:Director .
        ?x imdb:name ?nm . ?x imdb:birthDate ?bt }
```

and that *D* is:

```
CONSTRUCT { ?x rdf:type yago:FilmDirector.
            ?x foaf:name ?nm . ?x dbo:birthDate ?bt }
FROM <http://dbpedia.org>
WHERE {  ?x rdf:type yago:FilmDirector.
        ?x foaf:name ?nm . ?x dbo:birthDate ?bt }
```

Lastly, the user creates the linkset view definition ***f***=(*owl:sameAs*,*M*,*D*,π,μ) to materialize *owl:sameAs* links indicating that a director in *M* and a director in *D* are the same real-world object. As the match predicate, the user may choose the Levenshtein distance (Levenshtein, 1966) with threshold < 2 to compare the names of the directors and assume that the birthdates match only if they are equal. Then, for example, if views *M* and *D* respectively have two resources *u* and *v* with names “Tim Burton” and “Tim Button” and the same birthdate, then the linkset will have a triple (*u, owl:sameAs, v*), since the names have a Levenshtein distance of 1 (replace “t” by “r”), which satisfies the accepted threshold.

4 Incremental Maintenance of Linkset Views

4.1 Overview

Consider the *materialized linkset maintenance problem*, defined as follows: "Given two datasets, T and U, and a materialized linkset L from T to U, maintain L when updates on T or U occur".

A possible solution is to incrementally maintain L, that is, update L based on the updates on T or U. However, L does not contain the triples capturing the property values that generated the links. Hence, it is obviously impossible to detect when an update u on T or U affects L by just looking at the links in L. Thus, to incrementally maintain L, we propose a strategy that overcomes this lack of information by capturing the changes that must be applied to L using the information about the updates and the mappings of the catologue views adopted to define L.

Let V be a collection of catalogue views over T. Let u be an update on T and $\sigma_T(t_0)$ and $\sigma_T(t_1)$ be the states of T before and after u (the discussion is symmetric for updates on U). In the first process required by our incremental strategy, we need to capture the changes that affect each view v in $\boldsymbol{V}$ following four main steps:

1) Compute the set R^+ of resources in v affected by the inserted triples of u.
2) Compute the set R^- of resources in v that are affected by the deleted triples of u.
3) For s in $R^- \cup R^+$, retrieve the (new) property values of s from $\sigma_T(t_1)$, denoted P.
4) Associate R^- and P with the update timestamp t_u.

Let F be a catalogue view and $F \in \boldsymbol{V}$. Let $F[R^-(t_u)]$ be a collection of deleted resources of F associated with a given update timestamp t_u. Let $F[P(t_u)]$ be a collection of new property values of F associated with a given update timestamp t_u. Let t_1 be the current timestamp. Let $R^-[t_0,t_1]$ be the set of *accumulated deleted resources*, where $r \in R^-[t_0,t_1]$ iff $r \in F[R^-(t_u)]$ and $t_0 < r(t_u) < t_1$. Let $P[t_0,t_1]$ be the set of *accumulated property values*, where $p \in P[t_i,t_j]$ iff $p \in F[P(t_u)]$ and $t_i < p(t_u) < t_j$.

Suppose that L is a materialized linkset specified by the linkset view definition $\boldsymbol{l}=(p,F,G,\pi,\mu)$, where G is a catalogue view over U and $G[\sigma_U(t)]$ denote the set of triples that G returns when execute over state $\sigma_U(t)$ of U. In the second process of the incremental strategy, we incrementally update L following two main steps:

1) Delete from L all links whose subject or object occurs in $R^-[t_0,t_1]$.
2) Try to match $P[t_0,t_1]$ with the property values of a resource in $\sigma_U(t_1)$; if a match is found, add a link to L.

4.2 Normalization of View Mappings

Recall from Section 3.1 that the WHERE clause of a simple query F does not contain negations or the MINUS operator. In this section, we show how to transform the *triple patterns* of the WHERE clause of F into a *normalized form*, which simplifies the discussion in section 4.3.

Table 1 summarizes the allowed types of property paths in triple patterns (column 2) and the corresponding normalized form (column 3). Briefly, the *Normalization*

Process iteratively runs through the triple patterns of the WHERE clause and replaces their complex property paths by simpler ones until all paths are *predicate paths*, that is, paths of length one. Note that a property path generates one or more simpler triple patterns in a single *group graph pattern*, in the case of *Inverse Paths*, *Sequence Paths*, *Fixed Length Path* and *One or More Path* expressions. But a property path generates two simpler triple patterns in different *group graph patterns* with an UNION clause, in the case of *Alternative Path*, *Zero or More Path* and *Zero or One Path*. Additionally, a triple pattern marked with "(√)"needs **no further processing** to avoid a loop in the process. The original triple patterns are replaced by the normalized triple patterns in *F*. The output of the *Normalization Process* is a normalized view *F'* and a list of *predicate triple patterns*, denoted L_P, that is, triple patterns with predicate paths or predicate variables. Table 2 illustrates how the normalization works.

Table 1. Property Path Normalization.

Property Path	Original Triple Pattern	Normalized Triple Pattern
Predicate Path	?x *iri* ?y	do nothing
Inverse Path	?x ^*elt* ?y	?y *elt* ?x
Sequence Path	?x *elt1*/*elt2* ?y	?x *elt1* ?o1 . ?o1 *elt2* ?y
Alternative Path	?x *elt1*|*elt2* ?y	{ { ?x *elt1* ?y } UNION { ?x *elt2* ?y } }
Fixed Length Path (*n* > *0*) (*)	?x *elt{n}* ?y	?x *elt* ?o_1 . ?o_1 *elt* ?o_2 . … ?o_n *elt* ?y
One or More Path	?x *elt*+ ?y	?x *elt** ?o1 . (√) ?o1 *elt* ?o2 . ?o2 *elt** ?y . (√)
Zero or More Path	?x *elt** ?y	{ { ?x *elt** ?o1 . (√) ?o1 *elt* ?o2 . ?o2 *elt** ?y } (√) UNION { ?x *elt{0}* ?y } } (√)
Zero or One Path	?x elt? ?y	{ { ?x *elt* ?y } UNION { ?x *elt{0}* ?y } } (√)

(*) This syntactical form is not included in the specification of SPARQL 1.1, but it is supported by several triplestore systems.

Table 2. Example of the Normalization Process.

View Mapping

```
CONSTRUCT { ?x imdb:workedWith ?a }
WHERE { ?x ^imdb:director/(imdb:actress|imdb:actor) ?a }
```

Normalized Form

```
CONSTRUCT { ?x :workedWith ?a }
WHERE { ?o imdb:director ?x.
        {{ ?o imdb:actress ?a } UNION { ?o imdb:actor ?a }} }
```

L_P - List of Predicate Triple Patterns

```
[?o imdb:director ?x, ?o imdb:actress ?a, ?o imdb:actor ?a ]
```

4.3 Computing Affected Resources and New Property Values

We first summarize the notation to be used in what follows:

- T and U are datasets and u is an update on T
- $\sigma_T(t_i)$ is the state of T at time t_i, i=0,1
- F and G are catalogue view definitions over T and U, respectively
- $\sigma_F(t_i)=F[\sigma_T(t_i)]$ is the state of the view defined by view F at time t_i, i=0,1
- $\boldsymbol{l}=(p,F,G,\pi,\mu)$ is a linkset view definition over $\boldsymbol{v}$ and $\boldsymbol{w}$
- $\sigma_l(t_i)=\boldsymbol{l}[\sigma_T(t_i),\sigma_U(t_i)]$ is the state of $\boldsymbol{l}$ at t_i, i=0,1
- $\Delta^{-}_X(t_0,t_1) = \sigma_X(t_0) - \sigma_X(t_1)$ and $\Delta^{+}_X(t_0,t_1) = \sigma_X(t_1) - \sigma_X(t_0)$
  where X is either T, U, F, G or $\boldsymbol{l}$

A *deletion resources set query* of F for u, denoted F_u^{-}, is any query that computes a set of resources that contains the set of resources visible through F and affected by the deletions in u. Likewise, an *insertion resources set query* of F for u, denoted F_u^{+}, is any query that computes a set of resources that contains the set of resources visible through F and affected by the insertions in u. More precisely, we define:

- A *deletion resources set query of F for u*, denoted F_u^{-}, is a SPARQL query such that, for any states $\sigma_T(t_0)$ and $\sigma_T(t_1)$ of T such that $\sigma_T(t_0)$ and $\sigma_T(t_1)$ are the states before and after u, we have $\{ r \,/\, \exists p \exists o((r,p,o) \in \Delta_v^{-}(t_0,t_1) \} \subseteq F_u^{-}[(\sigma_T(t_0))]$
- An *insertion resources set query of F for u*, denoted F_u^{+}, is a SPARQL query such that, for any states $\sigma_T(t_0)$ and $\sigma_T(t_1)$ of T such that $\sigma_T(t_0)$ and $\sigma_T(t_1)$ are the states before and after u, we have $\{ r \,/\, \exists p \exists o((r,p,o) \in \Delta_v^{+}(t_0,t_1) \} \subseteq F_u^{+}[(\sigma_T(t_0))]$

Recall from Section 3.1 that we restrict view mappings to use the types of property paths in the second column of Table 1 and not to contain negations or the MINUS operator. This restriction has one important consequence, stated as follows.

A view mapping F over a dataset T is *monotonic* iff, for any two states $\sigma_T(t)$ and $\sigma_T(u)$, if $\sigma_T(t) \subseteq \sigma_T(u)$ then $F[\sigma_T(t)] \subseteq F[\sigma_T(u)]$.

Proposition 1: Assume that F is a view mapping whose WHERE clause uses the types of property paths listed in Table 1 and does not contain negations or the MINUS operator. Then, F is monotonic.

Monotonicity permits us to consider only deletions when constructing deletion resources set queries of F for u and, likewise, only insertions when constructing insertion resources set queries. Intuitively, if F were not monotonic, an insertion into $\sigma_T(t_0)$ might propagate to a deletion from $F[\sigma_T(t_0)]$ and, likewise, a deletion from $\sigma_T(t_0)$ might propagate to an insertion into $F[\sigma_T(t_0)]$.

Canonical Deletion and Insertion Resources Set. Let W_F be the WHERE clause and g_F be the graph in the FROM clause of F. Assume that F has already been normalized and let L_F be the set of predicate triple patterns that occur in W_F. Suppose that we materialize the set of deleted triples specified in the update u in state $\sigma_T(t_0)$ into a named graph g^{-}.

Assume that the predicate triple patterns in L_P are "a_k b_k c_k", for $k=1,\ldots,n$. Table 3 shows the template that generates the *canonical deletion resources set query* for F and u, denoted CF_u^-. Recall that the variable ?x identifies the resource of the catalogue view as defined in Section 3.1. Note that the results of CF_u^- are inserted into another named graph, denoted R^-, in which each resource is associated with the view identification and the timestamp of the update, denoted t_u. The template for the *canonical insertion resources set query* for F and u, denoted CF_u^+, is similarly defined, except that g^- is replaced by g^+, a named graph for the set of inserted triples u^+, and R^- is replaced by R^+, a named graph with the results of CF_u^+.We again resort to an example to illustrate the process of constructing CF_u^-. Table 3 recalls the definition of view M from Section 3.2, shows an update example, the query to populate g^- represented by the named graph <http://imdb.org/deletions> and finally the synthesized query CM_u^-.

Note that, if CM_u^- is executed before u is applied, the graph <http://imdb.org/data> has the necessary data to match the triple

```
(:Tim_Burton imdb:name 'Tim Burton').
```

Table 3. Example of the computation of affected resources.

Template of CF_u^-

```
INSERT { GRAPH <R¯> { ?x :view F . ?x :timestamp tu } }
WHERE { { GRAPH <g¯> { a1 b1 c1 } . GRAPH <gF> { WF } }
  UNION { GRAPH <g¯> { a2 b2 c2 } . GRAPH <gF> { WF } }
… UNION { GRAPH <g¯> { an bn cn } . GRAPH <gF> { WF } } }
```

***M* – the view mapping**

```
CONSTRUCT { ?x rdf:type yago:FilmDirector.
            ?x foaf:name ?nm . ?x dbo:birthDate ?bt }
WHERE { ?x rdf:type imdb:Director.
        ?x imdb:name ?nm. ?x imdb:birthDate ?bt}
```

u – the update

```
WITH <http://imdb.org/data>
DELETE DATA {:Tim_Burton
     imdb:name 'Tim Burton'}
```

Query to populate g^-

```
INSERT DATA {
GRAPH <http://imdb.org/deletions> {
:Tim_Burton imdb:name 'Tim
Burton'}}
```

L_P – `?x rdf:type imdb:Director, ?x imdb:name ?nm, ?x imd:birthDate ?bt`

CM_u^- – canonical deletion resources set query

```
INSERT { GRAPH <http://imdb.org/deletionsResources>
           { ?x :view M . ?x :timestamp "1" } }
WHERE{{GRAPH <http://imdb.org/deletions>{ ?x rdf:type imdb:Director}
       GRAPH <http://imdb.org/data> { ?x rdf:type imdb:Director .
             ?x imdb:name ?nm .?x imdb:birthDate ?bt }
 UNION{GRAPH <http://imdb.org/deletions> { ?x imdb:name ?nm }
       GRAPH <http://imdb.org/data> { ?x rdf:type imdb:Director .
             ?x imdb:name ?nm .?x imdb:birthDate ?bt } }
 UNION{ GRAPH <http://imdb.org/deletions> { ?x imdb:birthDate ?bt }
        GRAPH <http://imdb.org/data> { ?x rdf:type imdb:Director .
             ?x imdb:name ?nm .?x imdb:birthDate ?bt } } }
```

Indeed, returning to the general discussion, let $\sigma_{\mathbf{T}}(t_0)$ and $\sigma_{\mathbf{T}}(t_1)$ be the states of T before and after an update u is applied. We say that $CF_u^{-}[\sigma_{\mathbf{T}}(t_0)]$, the result of executing CF_u^{-} in state $\sigma_{\mathbf{T}}(t_0)$, is *the set of affected resources computed by* CF_u^{-} *in state* $\sigma_{\mathbf{T}}(t_0)$. Likewise, we say that $CF_u^{+}[\sigma_{\mathbf{T}}(t_1)]$, the result of executing CF_u^{+} in state $\sigma_{\mathbf{T}}(t_1)$, is *the set of affected resources computed by* CF_u^{+} *in state* $\sigma_{\mathbf{T}}(t_1)$. After $CF_u^{-}[\sigma_{\mathbf{T}}(t_0)]$ is computed, u can actually be applied and the triples in the named graph g^{-} can be cleared. However, CF_u^{+} has to be executed after u is applied, otherwise the state of T would not have the necessary data to match the triples in g^{+}.

To summarize, the process of computing the affected resources R^{-} and R^{+} follows four main steps: (1) intercept u and populate g^{+} and g; (2) execute F_u^{-}, populating R^{-}; (3) execute u; (4) execute F_u^{+}, populating R^{+}.

We stress that CF_u^{-} and CF_u^{+} are just a possible solution. Note that they can be synthesized at design time, right after the normalization, since the template will not change, only the data deleted or inserted. Furthermore, CF_u^{-} is correct in the following sense (a similar result holds for CF_u^{+}).

Proposition 2: Let F be a view mapping and u be an update over a dataset T. Let CF_u^{-} be the canonical deletion resources set query for F. Then, for any states $\sigma_{\mathbf{T}}(t_0)$ and $\sigma_{\mathbf{T}}(t_1)$ of T such that $\sigma_{\mathbf{T}}(t_0)$ and $\sigma_{\mathbf{T}}(t_1)$ are the states before and after u,

$$\{\ r\ /\ \exists \text{p} \exists \text{o}(\ (r,p,o) \in \Delta_{\mathbf{v}}^{-}(t_0,t_1)\ \} \subseteq CF_u^{-}[(\sigma_{\mathbf{T}}(t_0)].$$

New Property Values. After computing the graphs of the affected resources, we proceed to compute the named graph with the new property values, denoted P. Let W_F be the WHERE clause, C_F be the CONSTRUCT clause and g_F be the graph in the FROM clause of F. Table 4 shows the template and an example of the query to compute P.

Table 4. Computing the New Property Values.

Template:

```
INSERT { GRAPH <P> { CF . ?x :view F . ?x :timestamp tu } }
WHERE
{ { { SELECT DISTINCT ?d WHERE { GRAPH <R¯> { ?d ?p ?o } }
    UNION
    { SELECT DISTINCT ?i WHERE { GRAPH <R+> { ?i ?p ?o } } }
    GRAPH <gF> { WF }  FILTER ( ( ?x = ?i ) || ( ?x = ?d ) ) }
```

Example:

```
INSERT { GRAPH <http://imdb.org/newProperties>
          { ?x rdf:type yago:FilmDirector. ?x foaf:name ?nm .
            ?x dbo:birthDate ?dt . ?x :view M . ?x :timestamp "1" } }
WHERE
{ { { SELECT DISTINCT ?d
      WHERE{ GRAPH <http://imdb.org/deletedResources> { ?d ?p ?o } }
    UNION
    { SELECT DISTINCT ?i
      WHERE{ GRAPH <http://imdb.org/insertedResources> { ?i ?p ?o } } }
    GRAPH <http://imdb.org/data> { ?x rdf:type imdb:Director .
                              ?x imdb:name ?nm . ?x imdb:birthDate ?bt }
```

```
    FILTER ( ( ?x = ?i ) || ( ?x = ?d ) ) }
```

4.4 Updating a Materialized Linkset

Finally, the linkset can be updated according to the set of resources that were affected after the last timestamp maintenance. Let L be a materialized linkset, we first need to delete all links involving a resource in $R^{-}[t_0,t_1]$ according to the template of Table 5. Additionally, Table 5 shows an example of the linkset update process supposing that L was materialized in a named graph <http://linkset/directors>.

Table 5. Example of the Linkset Update Process.

Template	

`DELETE{ GRAPH <L> { ?s ?p ?o } }`
`WHERE { GRAPH <R¯> { ?s :view F. ?s :timestamp ?t. FILTER (?t > t0) }`
`        GRAPH <L>  { ?s ?p ?o } }`	

R^{-}**[t₀, t₁]**	R^{+}**[t₀, t₁]**
`imdb:Tim_Burton :view "M" .` `imdb:Tim_Burton :timestamp "1"`	`imdb:Ridley_Scott :view "M" .` `imdb:Ridley_Scott :timestamp "1"`
P	

`imdb:Tim_Burton rdf:type yago:FilmDirector .`
`imdb:Tim_Burton foaf:name "Tim Burton"; dbo:birthDate "1958-08-25" .`
`imdb:Tim_Burton :view "M" ; :timestamp "1" .`
`imdb:Ridley_Scott rdf:type yago:FilmDirector .`
`imdb:Ridley_Scott foaf:name "Ridley Scott";dbo:birthDate "1937-11-30".`
`imdb:Ridley_Scott :view "M" ; :timestamp "1"`	

Updating the Linkset	

`DELETE{ GRAPH <http://linkset/directors> { ?s ?p ?o } }`
`WHERE { GRAPH <http://imdb.org/deletionsResources>`
`          { ?s :view "M" . ?s :timestamp ?t . FILTER (?t > "0") }`
`        GRAPH <http://linkset/directors> { ?s ?p ?o } }`	

Old state of linkset *l*	**New state of linkset *l***
imdb:Tim_Burton :sameAs dbr:Tim_Burton	imdb:Ridley_Scott :sameAs dbr:Ridley_Scott

Then, the matching process is re-executed, using the triples in $P[t_0,t_1]$, instead of the whole view, and the new links are finally added to the materialized linkset.

We conclude this section with an observation about how the canonical queries are synthesized. We note that P also considers the resources in the deleted set R^{-} when computing the new property values. This is necessary since CF_u^{-} computes a superset R^{-} of the set of resources affected by deletions. That is, there might be a resource $r \in R^{-}$ that forced the deletion of a link of the form (r,p,o) from L, but r might not actually be affected by the deletions. Therefore, the algorithm has to recompute all such links. However, the problem of detecting the exact set of resources affected by deletions (or insertions) is NP-Complete, which is proved by a transformation from

Subgraph Isomorphism. Thus, synthesizing a deletion resources set query that returns the exact set of resources affected by a set of deletions is infeasible, unless P=NP.

5 Implementation and Evaluation

5.1 Architecture

The *Linkset Maintainer* tool implements the strategy detailed in Section 4. The tool was developed in the Java 7 programming language, using the Eclipse Luna IDE, JBoss Application Server 7 and ARQ API as the SPARQL Processor.

Fig. 2 summarizes the architecture of the tool. At initialization time, for each view *F* over a dataset *T*, the *View Controller* normalizes *F* and, at run time, it computes the set of affected resources and new property values of *F* with respect to updates submitted to *T*, as already discussed in sections 4.2 and 4.3. At initialization time, the *Linkset Controller* for a linkset ***l*** over views *F* and *G* registers itself with the *View Controllers* for *F* and *G* and computes the initial state of ***l***. At run time, it retrieves the sets of deleted resources and the sets of new property values of *F* and *G*, computes the accumulated set of deleted resources and the accumulated set of property values and incrementally maintains the linkset according to timestamp of the last maintenance, as discussed in section 4.4.

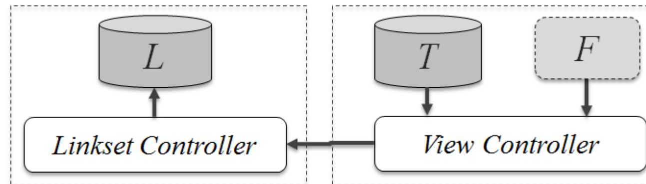

Fig. 2. Linkset View Maintainer Architecture.

The current implementation of the *Linkset Controller* uses Silk as the link discovery tool, since it provides an API that enables the matching process to be executed programmatically. The user only specifies the linkage rules and the tool automatically does the rest. The user may adopt other discovery tool, but s/he will have to manually manage the tool.

5.2 Evaluation Setup

In order to compare the performance of the incremental strategy with the full re-computation of linksets, we selected two datasets: an *IMDb* dump, with 44,855,096 triples, and the DBpedia endpoint, which at the time of the experiments had 883,644,235 triples. All experiments were executed on a computer with an Intel Core i5 1,7 GHz processor and 4 GB RAM, running OS X Yosemite 10.10.2.

We defined views about movie directors for each dataset. The view "IMDb_Director" has 41,929 resources and "DBpedia_Director" has 9,937 resources. Then, we materialized an owl:sameAs linkset of directors, using these views, by com-

paring their names and birth dates. The resulting linkset had 4,565 links. We performed updates on the *IMDb* dataset that affected view "IMDb_Director". All updates were similar to the following, except that they differ on the LIMIT clause to get an exact number of affected resources:

```
DELETE WHERE { ?s rdf:type imdb:Director } LIMIT 1
```

5.3 Experiments

We first compared the full rematerialization and the incremental strategy for the directors linkset in the presence of the updates described in Section 5.2. Fig. 3 shows the runtime of the updates, varying the number of affected resources. For each update, the runtime of the incremental strategy includes the time to compute the deleted resources and new property values, execute the update, and update the linkset. Likewise, the runtime of the full rematerialization includes the time to execute the update and rematerialize the linkset.

Since the queries to compute the new property values depend on the affected resources, after some point, it may become disadvantageous to use the incremental strategy. In the case of the directors linkset, this point was around 32K resources, which is 78% of the total number of resources of view "IMDb_Director", which has 41K resources. However, if the number of affected resources is small, incremental maintenance is far better than full rematerialization, as expected.

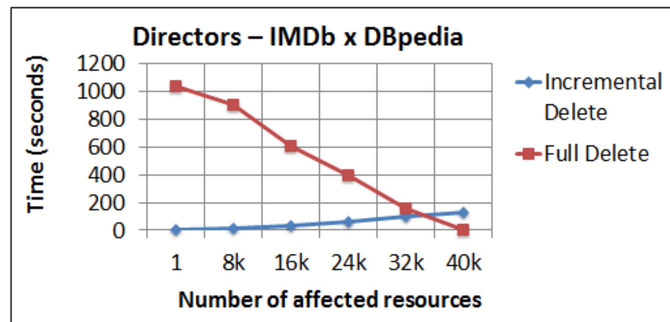

Fig. 3. Maintenance Performance of Linkset Directors.

We run a second experiment to assess, in a real-world situation, the percentage of resources, visible through views, that are affected by updates on a dataset. We based the experiments on the *DBpedia changesets*, that is, sets of changed triples extracted from Wikipedia, which are organized by year, month, day and hour and separated by the type of the update (added, removed, reinserted and cleared). For this second experiment, we defined two views over DBpedia about actors (49,308 resources) and actresses (7,309 resources), in addition to the directors' view. We then analyzed the number of view resources affected by the changesets from an entire day (April 28, 2015). We computed the number of updated resources by changeset and how many of these resources were visible through any of the views. Table 6 summarizes the results.

Considering each changeset as a single update, Table 6 shows an average of 99 resources per changeset, of which only 2% were visible through any of the views. Furthermore, the max number of affected resources in a single changeset was only 44. Therefore, this second experiment provides evidence that the number of affected resources tends to be small in real-world situations.

Table 6. Analysis of DBpedia Changesets.

	Total	Sets	Avg	Max
Updated Resources	551,236	5,568	99	975
View Resources	13,199	5,568	2	44

6 Conclusions

We first detailed an incremental strategy to keep linksets updated. We focused on how to compute the sets of affected resources that are visible through a view. Then, we showed how to keep the linksets updated based on such sets.

We presented the Linkset Maintainer, a tool that implements the incremental strategy. The tool was designed for an environment where it is possible to intercept the updates submitted to a dataset. However, the tool can be adapted to an environment where only the dataset changesets are available. In special, it is possible to adapt the strategy to compute the canonical deletion and insertion resources set, a crucial step of the process.

Based on the tool, we conducted experiments to measure the performance of both the incremental and the rematerialization strategies. The experiments demonstrated that the incremental strategy far outperforms full rematerialization, when the number of affected resources is relatively small, as expected. The results also showed that the runtime of the incremental strategy is negligible, when only a few resources are affected. We also analyzed DBpedia changesets from one day and concluded that, in the experiments, just a small percentage of the resources visible through the views were affected by updates. This experiment collected evidence that suggests that the incremental maintenance of materialized linksets will be efficient in practice, given that the number of resources that affects a view remains small.

As future work, we plan to continue the development of the tool to improve performance and to provide a better user interface to help the definition of views and linksets. Finally, we plan to make the tool freely available.

Acknowledgments

This work was partly funded by CNPq under grants 153908/2015-7, 557128/2009-9, 444976/2014-0, 303332/2013-1, 442338/2014-7 and 248743/2013-9 and by FAPERJ under grants e E-26-170028/2008 and E-26/201.337/2014.

References

1. Berners-Lee, T.: Linked Data, http://www.w3.org/DesignIssues/LinkedData.html (2006)
2. Bouza, A.: The Movie Ontology, http://www.movieontology.org (2010)
3. Casanova, M.A., Vidal, V.M.P., Lopes, G.R., Leme, L.A.P.P., Ruback, L.: On Materialized sameAs Linksets. In: Database and Expert Systems Applications, pp. 377-384. Springer International Publishing (2014)
4. Endris, K.M., Faisal, S., Orlandi, F., Auer, S., Scerri, S.: Interest-based RDF update propagation. In: The Semantic Web-ISWC 2015, pp. 513-529. Springer International Publishing (2015)
5. Gupta, A., Mumick, I., Subrahmanian, V.: Maintaining Views Incrementally. In: ACM SIGMOD Record, pp. 157– 166 (1993)
6. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language, http://www.w3.org/TR/sparql11-query/ (2013)
7. Hung, E.; Deng, Y.; Subrahmanian, V.: Maintaining RDF views. In: Tech. Rep CS-TR-4612 (UMIACS-TR-2004-54), University of Maryland (2004)
8. Ibáñez, L. D., Skaf-Molli, H., Molli, P., & Corby, O.: Col-graph: Towards writable and scalable linked open data. In: The Semantic Web–ISWC 2014, pp. 325-340. Springer International Publishing (2014)
9. Levenshtein, V.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet Physics Doklady, volume 10 (1966)
10. Ngomo, A., Auer, S.: LIMES - A Time-Efficient Approach for Large-Scale Link Discovery on the Web of Data. In: Proc. IJCAI 2011, pp. 2312–2317 (2011)
11. Popitsch, N., Haslhofer, B.: DSNotify – A Solution for event detection and link maintenance in dynamic triplesets. In: Journal of Web Semantics, 9(3), pp. 266–283 (2011)
12. Staudt, M., Jarke, M.: Incremental maintenance of externally materialized views. In: Proc. VLDB 1996, pp. 75–86 (1996)
13. Vidal, V.M.P., Casanova, M.A., Cardoso, D.S.: Incremental Maintenance of RDF Views of Relational Data.In: Proc. ODBASE 2013, pp 572-587 (2013)
14. Vidal, V. *et al*.: Specification and Incremental Maintenance of Linked Data Mashup Views. In: Advanced Information Systems Engineering, pp. 214-229. Springer International Publishing (2015)
15. Volz, J., Bizer, C., Gaedke, M.: Web of Data Link Maintenance Protocol - Maintaining Links Between Changing Linked Data Sources, http://www4.wiwiss.fu-berlin.de/bizer/silk/wodlmp (2009a)
16. Volz, J., Bizer, C., Gaedke, M., Kobilarov, G.: Discovering and Maintaining Links on the Web of Data. Proc. ISWC 2009, pp. 650-665 (2009b)