Materialized sameAs Link Maintenance with Views

Elisa Souza Menendez

Dissertação (Mestrado em Informática). Pontifícia Universidade Católica do Rio de Janeiro. Rio de Janeiro, 2015.

Pontifícia Universidade Católica
do Rio de Janeiro

Elisa Souza Menendez

Materialized sameAs Link Maintenance with Views

Dissertação de Mestrado

Dissertation presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática, PUC-Rio, as partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Marco Antonio Casanova

Rio de Janeiro
July 2015

PUC-Rio - Certificação Digital Nº 1321832/CA

Pontifícia Universidade Católica
do Rio de Janeiro

Elisa Souza Menendez

Materialized sameAs Link Maintenance with Views

Dissertation presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requirements for the degree of Mestre.

Prof. Marco Antonio Casanova
Advisor
Departamento de Informática – PUC-Rio

Prof. Giseli Rabello Lopes
Departamento de Ciência da Computação – UFRJ

Prof. Vânia Maria Ponte Vidal
Departamento de Computação – UFC

Prof. José Antonio Fernandes de Macêdo
Departamento de Computação – UFC

Prof. José Eugênio Leal
Coordenador Setorial do Centro Técnico Científico – PUC-Rio

Rio de Janeiro, July 20th, 2015

PUC-Rio - Certificação Digital Nº 1321832/CA

All rights reserved

Elisa Souza Menendez

Graduated in Information Systems from Federal University of Sergipe (UFS), São Cristóvão - Brazil in 2013. She joined the Master in Informatics at Pontifical Catholic University of Rio de Janeiro (PUC-Rio) in 2013.

Bibliographic data

Menendez, Elisa Souza

Materialized sameAs link maintenance with views / Elisa Souza Menendez ; advisor: Marco Antonio Casanova. – 2015.

68 f. : il. (color) ; 30 cm

Dissertação (Mestrado em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2015.

Inclui bibliografia

1. Informática – Teses. 2. Interligações sameAs. 3. Manutenção de interligações. 4. Dados interligados. 5. Atualizações de visões. I. Casanova, Marco Antonio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

PUC-Rio - Certificação Digital Nº 1321832/CA

Acknowledgments

I would like to say a special thank you to my parents, Gracinha and Angel, for their support and encouragement during all these years of study. To all my family from Nikiti city, especially my aunt Arlete, who gave me a home in her house.

To Marco Antonio Casanova, the best advisor I could ever ask for. I hope someday, a student can admire me as much as I admire him.

To PUC-Rio and CAPES for funding my research.

To all my classmates, professors and staff from the Informatics Department. Thanks for all your help and for always being so accommodating.

PUC-Rio - Certificação Digital Nº 1321832/CA

Abstract

Menendez, Elisa Souza; Casanova, Marco Antonio (Advisor). **Materialized sameAs Link Maintenance with Views.** Rio de Janeiro, 2015. 68p. MSc. Dissertation – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

In the Linked Data field, data publishers frequently materialize sameAs links between two different datasets using link discovery tools. However, it may be difficult to specify linking conditions, if the datasets have complex models. A possible solution lies in stimulating dataset administrators to publish simple predefined views to work as resource catalogues. A second problem is related to maintaining materialized sameAs linksets, when the source datasets are updated. To help solve this second problem, this work presents a framework for maintaining views and linksets using an incremental strategy. The key idea is to re-compute only the set of updated resources that are part of the view. This work also describes an experiment to compare the performance of the incremental strategy with the full re-computation of views and linksets.

PUC-Rio - Certificação Digital Nº 1321832/CA

Keywords

sameAs Links; Link Maintenance; Linked Data; View Update

Resumo

Menendez, Elisa Souza; Casanova, Marco Antonio. **Manutenção de Links sameAs Materializados utilizando Visões.** Rio de Janeiro, 2015. 68p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Na área de dados interligados, usuários frequentemente utilizam ferramentas de descoberta de links para materializar links sameAs entre diferentes base de dados. No entanto, pode ser difícil especificar as regras de ligação nas ferramentas, se as bases de dados tiverem modelos complexos. Uma possível solução para esse problema seria estimular os administradores das base de dados a publicarem visões simples, que funcionem como catálogos de recursos. Uma vez que os links estão materializados, um segundo problema que surge é como manter esses links atualizados quando as bases de dados são atualizadas. Para ajudar a resolver o segundo problema, este trabalho apresenta um framework para a manutenção de visões e links materializados, utilizando uma estratégia incremental. A ideia principal da estratégia é recomputar apenas os links dos recursos que foram atualizadas e que fazem parte da visão. Esse trabalho também apresenta um experimento para comparar a performance da estratégia incremental com a recomputação total das visões e dos links materializados.

PUC-Rio - Certificação Digital Nº 1321832/CA

Palavras-chave

Links sameAs, Manutenção de Links, Dados interligados, Atualização de Visões

Table of Contents

PUC-Rio - Certificação Digital Nº 1321832/CA

PUC-Rio - Certificação Digital Nº 1321832/CA

List of Figures

PUC-Rio - Certificação Digital Nº 1321832/CA

List of Tables

PUC-Rio - Certificação Digital Nº 1321832/CA

1
Introduction

1.1 Motivation

The Linked Data initiative (BERNERS-LEE, 2006) defines best practices for publishing data on the Web, using RDF triples to connect and structure it. The idea became popular and the number of triples grew significantly, but there was a concern about the lack of links between different datasets.

Link discovery tools, such as LIMES (NGOMO; AUER, 2011) and Silk (VOLZ et al., 2009), came as solutions to help create and materialize linksets, that is, to explicitly store the set of links. These tools, however, are semi-automatic, since users have to set linkage rules, that is, they have to specify conditions that resources must fulfill to be interlinked. Data publishers also have to specify which type of RDF link should be created. The most common one is the sameAs link, which has the form (*s*, *owl:sameAs*, *o*) and indicates that *s* and *o* denote the same resource.

Defining the linkage rules can be a complex task, since the user must know how the datasets are modeled to specify the conditions. Thus, this work presents a strategy to deal with this problem, in which the key idea is to use SPARQL-based views defined by the administrator of each dataset. The views should act as resource catalogues, that is, sets of resources with useful properties. Hence, the user who wants to create a materialized linkset only selects two of the pre-defined *catalogue views* and performs a simpler post-configuration.

Since datasets are continually updated, link maintenance is another problem of the Linked Data field. For instance, when a remote resource used in a link is removed, the link is invalidated and should also be removed. To address such problem, Casanova *et al*. (2014) proposed an incremental strategy to keep sameAs linksets updated, similar to the traditional incremental view maintenance strategies.

PUC-Rio - Certificação Digital Nº 1321832/CA

1.2 Goal and Contributions

The main contributions of this work are to improve, implement and evaluate the solution proposed by Casanova *et al.* (2014).

The main improvement in the solution is the extension of the catalogue view definition to allow administrators to define their views with more flexibility. Therewith, we also introduce the process of normalizing SPARQL queries, simplifying complex SPARQL elements.

We also describe the implementation of the proposed architecture with Master Controllers, View Controllers and Linkset Controllers in the Linkset Maintainer Tool.

Finally, we describe experiments to test the performance of the incremental strategy and to compare this strategy with a linkset re-computation basic strategy.

1.3 Dissertation Structure

This dissertation is structured as follows. Chapter 2 presents the basic concepts and summarizes related work. Chapter 3 describes the concept of linkset views. Chapter 4 presents the incremental strategy and the Linkset Maintainer tool. Chapter 5 covers the evaluation and results of the tool. Finally, Chapter 6 presents the conclusions and proposes future work.

PUC-Rio - Certificação Digital Nº 1321832/CA

2
Background

This chapter provides an overview of the main concepts related to this dissertation. Section 2.1 introduces key definitions about Linked Data, its relation with the Resource Description Framework (RDF) and how it contributed to the formation and growth of the Web of Data. Section 2.2 covers the SPARQL Query Language, especially the latest features introduced in Version 1.1: property paths and updates. Finally, Section 2.3 describes related work, divided into link discovery tools, link maintenance tools and view maintenance strategies.

PUC-Rio - Certificação Digital Nº 1321832/CA

2.1 Linked Data

Tim Bernes-Lee introduced a set of best practices for publishing and interlinking structured data on the Web, known as Linked Data (BERNERS-LEE, 2006). There are four main principles that define Linked Data:

- Use URIs as names for things.
- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide useful information using the standards (RDF, SPARQL).
- Include links to other URIs, so that they can discover more things.

The idea of the first principle is to extend the classic Web and use URIs (Uniform Resource Identifiers) to identify not only documents, but also any object or concept of the real world. URIs can identify concrete things, such as people, places, and cars, or abstract concepts, such as feelings and relations (HEATH; BIZER, 2011).

Once there is an URI defining something, it needs to be combined with the HTTP protocol in order to enable the URI to be *dereferenced*, that is, to provide access to the description of objects and concepts.

The third principle promotes the use of standard content format to enable different applications to process Web content. The structured data can be represented and shared using a simple graph-based model, known as RDF (Resource Description Framework), described in section 2.1.1.

Finally, the fourth principle is considered the most important for the scope of this work. This principle promotes the use of RDF triples to describe relationships between resources. Such triples are often referred to as *links*. For instance, to connect a person with a place, one may use the relationship "works". Moreover, links should also be created between different datasets in order to create a global data space, called the Web of Data, described in section 2.1.2.

2.1.1. Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a framework for expressing information about resources (MANOLA; MILLER, 2014). Resources can be anything (documents, people, objects, concepts, etc.) and are described using triples. A *triple* is a statement that has a subject, a predicate and an object. Informally, an instance of a statement can be "The Mona Lisa was created by Leonardo Da Vinci", in which the subject is "The Mona Lisa", the predicate is "was created by" and the object is "Leonardo Da Vinci". The combination of the statements forms a graph, as shown in Figure **1**.

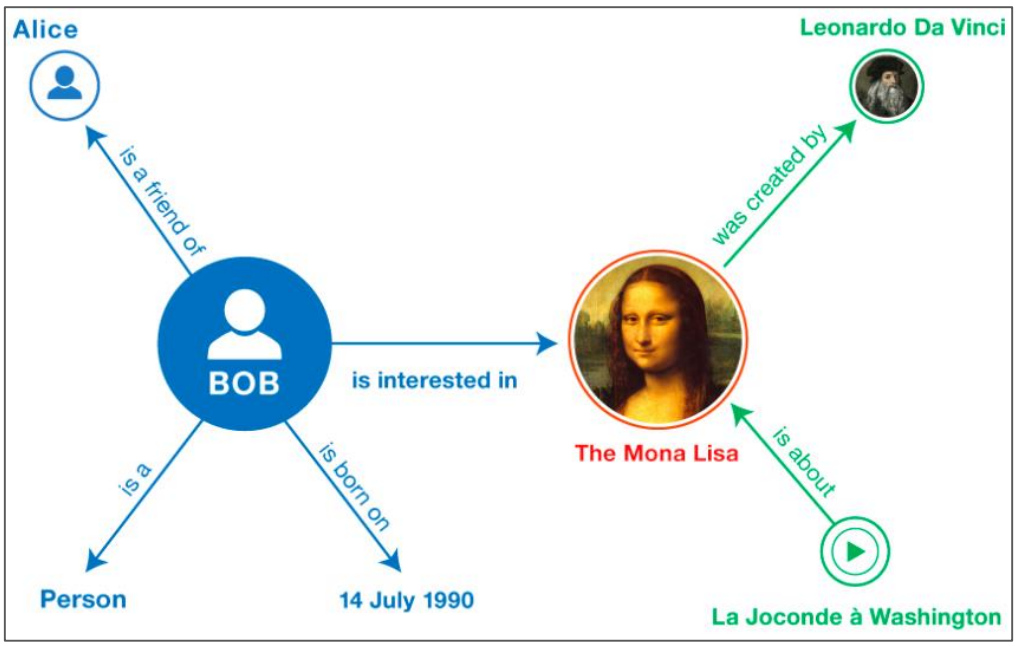

Figure 1 - Informal RDF Graph (MANOLA; MILLER, 2014)

PUC-Rio - Certificação Digital Nº 1321832/CA

Formally, in RDF, the subject and the predicate of the triple have to be represented as an URI, and the object can be an URI or a literal. URI stands for "Uniform Resource Identifier" and is a global identifier that allows different people to reuse the URI to identify the same thing. For instance, the dataset DBpedia uses the URI `http://dbpedia.org/resource/Mona_Lisa` to denote the Mona Lisa painting described by the corresponding Wikipedia article. Additionally, DBpedia uses the URI `http://dbpedia.org/ontology/author` to represent the predicate "was created by" and the URI `http://dbpedia.org/resource/Leonardo_da_Vinci` to represent the object "Leonardo Da Vinci". In turn, a literal is a basic value that is not an URI. For instance, DBpedia denotes the following triple, in which the object is literal:

```
(http://dbpedia.org/resource/Mona_Lisa,
      http://dbpedia.org/property/otherTitle,
           "La Joconde")
```

In practice, RDF is used in combination with vocabularies that provide semantic information about the resources. Examples of popular vocabularies are:

- RDF Schema: defines the basic idea of classes and properties. For example, one can state that the URI `http://www.example.org/friendOf` can be used as a property and that the subjects and objects of this predicate must be resources of class `http://www.example.org/Person`. Then, one can say that the resources Bob and Mary are of the type Person, and that Bob is a friend of Mary.
- OWL (Web Ontology Language): extends the expressivity of RDF Schema with additional primitives, such as *equivalent class*, *equivalent property*, *different of*, *same as*, etc.
- FOAF (Friend of a Friend): describes people, their activities and their relations to other people.
- Dublin Core: defines general attributes such as *title*, *creator*, *date* and *subject.*

2.1.2. Web of Data

The Web of Data forms a large global graph connecting RDF datasets from all sorts of topics, such as locations, people, publications, music, movie, and etc. The idea of the Web of Data started to gain force in 2007 with the *Linked Open Data*

PUC-Rio - Certificação Digital Nº 1321832/CA

(LOD)[1] project. The aim of this project was to identify existing datasets available under open licenses and to publish them in RDF, according to the Linked Data Principles (Heath and Bizer, 2011). Subsequently, several individuals and organizations were stimulated to publish their data in the LOD using the Linked Data principles. Figure 2 shows the LOD graph for the datasets published until August 2014.

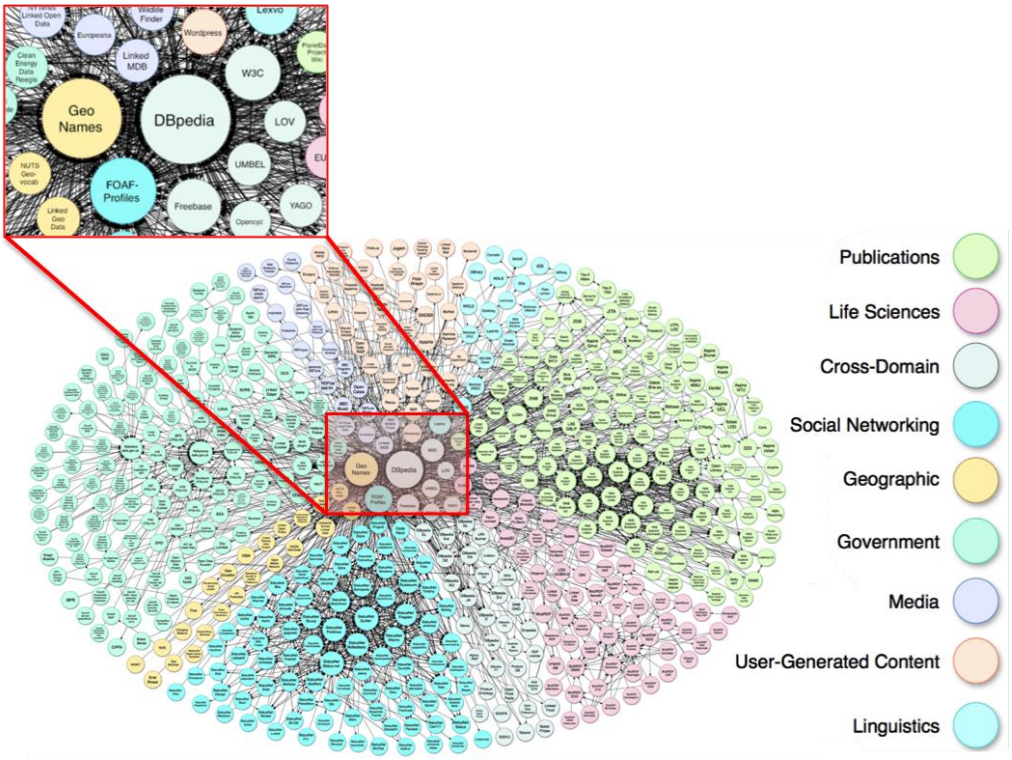

Figure 2 - The LOD Cloud Diagram on August 2014

The LOD has some interesting statistics[2], for example, as of August 2014, the total number of datasets was 1004; the Social Networking domain was 51.28% of the total; 56.11% of the crawled datasets link to at least one other dataset, and the remaining datasets are only targets of RDF links. In total, 23.17% datasets use proprietary vocabularies, while nearly all (99.87%) datasets use non-proprietary vocabularies. A vocabulary is *non-proprietary* if there are at least two datasets using the vocabulary.

Figure 3 shows the three predicates most used for interlinking, by category. It is important to highlight that the link *owl:sameAs*, which denotes that a resource is the same as other resource, appears in the list of 7 out of the 8 categories, and is the first of the list in 6 categories. Thus, the *owl:sameAs* is the most frequent

[1] http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData
[2] http://linkeddatacatalog.dws.informatik.uni-mannheim.de/state/

PUC-Rio - Certificação Digital Nº 1321832/CA

predicate for interlinking in the LOD.

Category	Predicate	Usage	Category	Predicate	Usage
social web	foaf:knows	60.27%	life sciences	owl:sameAs	52.17%
	foaf:based_near	35.69%		rdfs:seeAlso	48.48%
	sioc:follows	34.34%		dct:creator	21.74%
publications	owl:sameAs	32.20%	government	dct:publisher	47.57%
	dct:language	25.42%		dct:spatial	30.10%
	rdfs:seeAlso	23.73%		owl:sameAs	24.27%
user-generated content	owl:sameAs	53.13%	geographic	owl:sameAs	64.29%
	rdfs:seeAlso	21.88%		skos:exactMatch	21.43%
	dct:source	18.75%		skos:closeMatch	21.43%
media	owl:sameAs	81.25%	cross-domain	owl:sameAs	80.00%
	rdfs:seeAlso	18.75%		rdfs:seeAlso	52.00%
	foaf:based near	18.75%		dct:creator	20.00%

Figure 3 - The three most used predicates for interlinking, by category

In order to create links between different datasets, one can use popular identifiers, such as International Standard Book Number (ISBN), DOI (Digital Object Identifier), person's ID number, etc. However, in some cases, different datasets do not share a common identifier and need to be linked based on the similarity between two resources. Several tools were developed to help the task of finding links between different datasets and contribute to the expansion of the Web of Data.

2.2 SPARQL Query Language

The SPARQL query language (HARRIS; SEABORNE, 2013) can be used to express queries over RDF graphs. A simple example of a SPARQL query is shown below, in which the result is all the triples that have foaf:Person as its type.

```
SELECT ?subject
WHERE { ?subject rdf:type foaf:Person }
```

The `SELECT` clause identifies the variables that will appear in the result (in this case, `?subject`). The `WHERE` clause contains the graph pattern that is matched with a RDF graph. The pattern in this example is a single triple, but SPARQL also supports aggregation, subqueries, negation, filters, and etc.

PUC-Rio - Certificação Digital Nº 1321832/CA

Another way to express queries in SPARQL is using the `CONSTRUCT` query form, which returns in a single RDF graph specified by a graph template. In the example below, the new graph would contain all the resources with the property foaf:givenName replaced by the property foaf:name.

```
CONSTRUCT { ?subject foaf:name ?name
WHERE { ?subject foaf:givenName ?name }
```

2.2.1. Property Paths

One of the features of the latest version, SPARQL 1.1, is the support for *property paths*, which is a possible path between two nodes in a graph. Table 1 shows the syntax of the *property paths*.

Syntax Form	***Expression Name***	***Matches***
iri	*Predicate Path*	An URI. A path of length one.
^*elt*	*Inverse Path*	Inverse path (object to subject).
elt1 / *elt2*	*Sequence Path*	A sequence path of *elt1* followed by *elt2*.
elt1 | *elt2*	*Alternative Path*	An alternative path of *elt1* or *elt2* (all possibilities are tried).
*elt**	*Zero or More Path*	A path that connects the subject and object of the path by zero or more matches of *elt*.
elt+	*One or More Path*	A path that connects the subject and object of the path by one or more matches of *elt*.
elt?	*Zero or One Path*	A path that connects the subject and object of the path by zero or one matches of *elt*.
elt{n}[*]	*Fixed Length Path*	A path that connects the subject and the object of the path by exactly *n* matches of *elt*.
!*elt*	*Negated Path*	Every match that is not *elt.*
(*elt*)	*Group Path*	A group path *elt*, where brackets control precedence.

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

Table 1 - Property Path Syntax

As an example of the *Sequence Path* expression, the query below finds the name of any people that "Alice Smith" knows.

```
SELECT  ?name
```

PUC-Rio - Certificação Digital Nº 1321832/CA

```
WHERE { ?subject foaf:firstName "Alice" .
        ?subject foaf:lastName  "Smith" .
        ?subject foaf:knows/foaf:name  ?name }
```

Note that this *Sequence Path* could be replaced by:

```
?subject foaf:knows ?person .
?person foaf:name ?name
```

The *Inverse Path* expression reverses the direction of the predicate, swapping the roles of subject and object, as in:

```
SELECT  ?subject
WHERE { "Alice" ^foaf:firstName ?subject }
```

As for the *Zero or More Path* expression, the following query returns all types and supertypes of the resources:

```
SELECT  ?subject ?type
WHERE { ?subject rdf:type/rdfs:subClassOf* ?type }
```

2.2.2. Updates

Another important feature of SPARQL 1.1 is the possibility to update RDF datasets, and to insert or delete triples. There are three types of operations: *update data*, *delete where*, and *modify*. The *update data* inserts or deletes triples given inline in the request, for example:

```
INSERT DATA {<http://example/book1> dc:title "A new book" }
```

The *modify* operation can be used to remove or add triples based on bindings for a query pattern specified in a `WHERE` clause, as in:

```
WITH <http://example/addresses>
DELETE { ?person foaf:firstName "Bill" }
INSERT { ?person foaf:firstName "William" }
WHERE  { ?person foaf:firstName "Bill" .
         ?person foaf:lastName "Smith" }
```

Finally, the *delete where* operation is a shortcut for *modify* (delete), in which bindings matched by the `WHERE` clause are used to define the triples that will be deleted. Despite that, the *delete where* is more limited than *modify*, since it is not possible to use filters or property paths. An example is:

```
DELETE WHERE { ?person foaf:firstName "Fred" .
     ?person ?property ?value }
```

PUC-Rio - Certificação Digital Nº 1321832/CA

2.3 Related Work

2.3.1. Link Discovery Tools

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. Figure 4 shows the Silk Workbench, in which the user can define the linkage rules by setting the properties that will be compared (e.g. *foaf:name, rdfs:label*), the transformations (e.g. Lower case function) that will be applied and the similarity measures (e.g. Levenshtein distance) to compare the values.

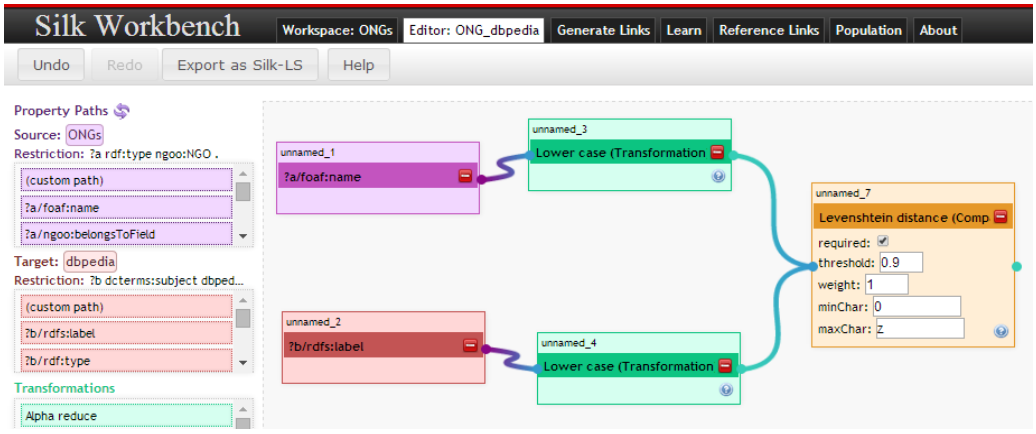

Figure 4 - Silk Workbench: defining linkage rules.

In addition to the link discovery engine, Silk has another component for evaluating the links generated. The framework provides a Web interface for users to evaluate the correctness and completeness of the generated links. They can set the linkage rules and submit triples that have an expected result. Afterwards, the tool shows the exact values of the metrics and aggregations, so that the user can check if it worked as expected and fine-tune the linkage rules, if necessary.

2.3.2. Link Maintenance Tools

The authors of Silk also proposed a protocol for link maintenance, called the Web of Data – Link Maintenance Protocol (WOD-LMP), to deal with the changes that

PUC-Rio - Certificação Digital Nº 1321832/CA

may occur in datasets (VOLZ *et al*., 2009; VOLZ; BIZER; GAEDKE, 2009). The protocol covers three use cases:

- Link Transfer to Target – the source sends notifications to the target when a link is created or deleted.
- Request of Target Change List – the source requests to the target a list of changes in a specified time range.
- Subscription of Target Changes – the source sends the link notifications, and the target saves this information to further notify the source about changes in the selected resources.

DSNotify is another tool that supports link maintenance (POPITSCH; HASLHOFER, 2011). The tool can be described as a general-purpose change detection framework that notifies linked data sources about events (create, remove, move, and update) in their remote resources. To deal with these changes, DSNotify uses a specific OWL Lite vocabulary called DSNotify Eventset Vocabulary, which allows a detailed description (what, how, when, and why) of the events.

PUC-Rio - Certificação Digital Nº 1321832/CA

2.3.3. View Maintenance Strategies

This dissertation is also related to strategies for materialized view maintenance. In 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; MUMICK; SUBRAHMANIAN, 1993; STAUD; JARKE, 1996). This strategy was adapted to maintain RDF views of the underlying relational data (VIDAL, CASANOVA; CARDOSO, 2013). Both contexts showed that 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.

This dissertation is also closely related to strategies designed for maintaining RDF views over RDF datasets (HUNG; DENG; SUBRAHMANIAN, 2004; VIDAL *et al*., 2015), 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.

Additionally, the proprietary systems that support the incremental maintenance of views, such as Oracle RDF Store[3], 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 were a solution for the maintenance of SPARQL-based views in the literature, we would still not be able to directly use it.

As already mentioned in the introduction, the work reported in this paper differs from the work of 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 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 that has been neglected in the literature.

PUC-Rio - Certificação Digital Nº 1321832/CA

[3] http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/rdfsemantic-graph-1902016.html

3
Linkset Views

This chapter presents the main concepts about Linkset Views and how they can help the creation process of materialized links. Section 3.1 starts with formal notations about Linked Data, Views and Linkset Views. A key point of this section is the definition of simple property path queries, along with a running example to illustrate the notation and to provide further explanations. Section 3.2 shows how data publishers can benefit from views created by the administrators of the datasets in the definition process of Linkset Views.

PUC-Rio - Certificação Digital Nº 1321832/CA

3.1 Notation and Example

3.1.1. Basic Linked Data Notation

Linked Data has some basic concepts that need to be formally described for the purpose of this work.

An *RDF dataset T* is a set of RDF triples (Harris and Seaborne, 2013). A *triple* (*s, p, v*) in *T* defines a property *p* of a resource *s*.

Let *U* be a second dataset. A *link* from *T* to *U* is an RDF triple *(s, p, o)*, such that *s* is defined in *T*, *o* is defined in *U* and *p* is not *rdf:type* (Alexander et al., 2011). The set of links from *T* to *U* is a *linkset* from *T* to *U*. A *sameAs link* is a link of the form *(s, owl:sameAs, o)*, which asserts that *s* denotes the same resource as *o*.

The similarity measure σ is used to compare two objects and determine how similar they are. Formally, a function $\sigma : (D_1 \times \ldots \times D_n) \times (D_1 \times \ldots \times D_n) \rightarrow \mathcal{R}$ is a similarity measure for tuples in $D_1 \times \ldots \times D_n$ iff, for any $x,y \in D_1 \times \ldots \times D_n$, $\sigma(x,y) \geq 0$, $\sigma(x,x) \geq \sigma(x,y)$ and $\sigma(x,y) = \sigma(y,x)$ (Euzenat and Shvaiko, 2007).

3.1.2. Views and Linkset Views Notation

Other concepts that need to be formally described are *catalogue views* and *linkset views*, since they will be used in this work. For that purpose, this section will introduce an abstract notation based on a minimum set of simple SPARQL 1.1 constructs (Harris and Seaborne, 2013).

A *simple construct query* can be understood as a catalogue of resources of a given type with properties defined by SPARQL 1.1. Such catalogues will be used to generate sameAs linksets. More precisely, a SPARQL query *F* is a *simple construct query*, or a *simple query*, iff:

- 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 subjected to certain restrictions, as will be explained in Section 4.4.

A *catalogue view definition* is a pair $v = (V_F, F)$, where *F* is a simple query, called the *view mapping*. The *view vocabulary* V_F is the vocabulary of *F* and consists of a single class and an ordered list of properties. When there is no need to highlight the view vocabulary, we will simply refer to *F* as the view definition.

The *materialization of v* is the process of computing the set of triples that *F* returns when execute over state $\sigma_T(t)$ of *T*, denoted $F[\sigma_T(t)]$, and explicitly storing it as part of a dataset.

Finally, a *linkset view* is a quintuple $l = (p, F, G, \pi, \mu)$, where:

- *p* is an object property;
- *F* and *G* are simple queries, whose vocabularies have the same cardinality *n;*
- π is a permutation of *(1,...,n)*, called the *alignment* of *l;*
- μ is a *2n*-relation, called the *match predicate* of *l.*

PUC-Rio - Certificação Digital Nº 1321832/CA

A linkset view could also be defined as $l = (p, v, w, \pi, \mu)$, where $v=(V_F, F)$ and $w=(V_G, G)$ are catalogue views. If $\boldsymbol{v}$ and $\boldsymbol{w}$ are replaced by their queries, F and G, respectively, it returns to the previous form.

Let $V_F=\{C,P_1,\ldots,P_n\}$ be the vocabulary of F and $V_G=\{D,Q_1,\ldots,Q_n\}$ be the vocabulary of G. Intuitively, the alignment π in l indicates that, for each $k=1,\ldots,n$, the match predicate will compare values of P_k with values of Q_m, where $m = \pi(k)$. The notion of alignment can be generalized to permit more sophisticated alignments and mappings, such as mapping the concatenation of last name and first name into a single name.

Let T be the dataset specified in the FROM clause of F, and U be the dataset specified in the FROM clause of G. The linkset view definition l induces a set of triples from T to U, denoted $l[T,U]$, as follows:

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

The *materialization* of l is the process of computing the set $l[T,U]$ and explicitly storing it as part of a dataset.

Although these definitions are general, we stress at this point that the rest of this dissertation will address only linkset views where p is the owl:sameAs property (http://www.w3.org/2002/07/owl#sameAs).

3.1.3. Example

The notation previously introduced can be better understood with examples. Thus, this section presents two datasets and their respective ontologies that will support a linkset view example.

The *Lattes dataset* (*BrCV*) represents CVs of Brazilian researchers and was extracted from the Lattes platform. Suppose that *BrCV* has a fictitious SPARQL endpoint "http://lattes.br/sparql" and uses the *Lattes* ontology - a fragment of this is presented in Figure 5. The fictitious namespace for the ontology is http://onto.lattes.br/ and its prefix is "la:".

PUC-Rio - Certificação Digital Nº 1321832/CA

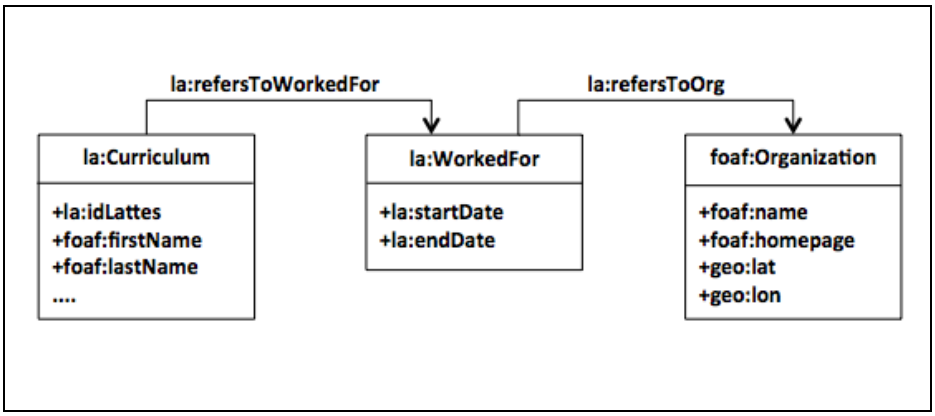

Figure 5 - A simplified fragment of the *Lattes Ontology*

- The *Semantic Web Conference Corpus dataset* (*SWCC*) contains triples about the main conferences and workshops in the area of Semantic Web research. Its fictitious SPARQL endpoint is http://semanticweb.org/sparql, and the ontology used is the *Semantic Web Conference (SWC) ontology* (Möller et al., 2009). A fragment of *SWC* is shown in Figure **6**. The fictitious namespace is http://data.semanticweb.org/ns/swc/ontology# and its prefix is "swc:".

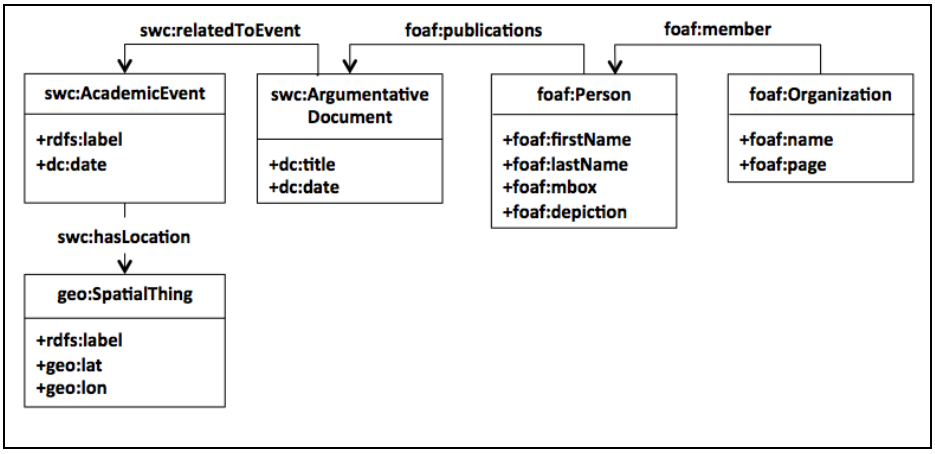

Figure 6 - A simplified fragment of the *Semantic Web Conference Ontology*

Now, suppose that a user wants to link researchers in the *SWCC* dataset with those represented in the *BrCV* dataset by their CVs. He/She could compare the person's name from both datasets and, to disambiguate, use the homepage of the organization the person works for.

Thereby, the following queries exemplify two simple property path queries, as defined in the previous section. The first query, named F_{SWCC}, is evaluated over the *Semantic Web Conference Corpus* dataset and is defined as follows:

PUC-Rio - Certificação Digital Nº 1321832/CA

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

Note that the CONSTRUCT clause of F_{SWCC} selects, for a *person ?x*, the values of properties *foaf:firstName* and *foaf:lastName*, and the value of property *foaf:page* of the organization related to ?x by the inverse property *foaf:member*.

The second query, named G_{BrCV}, is evaluated over the Lattes dataset and is defined as follows:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

Note that the CONSTRUCT clause of G_{BrCV} selects, for a person ?x, the values of properties *foaf:firstName* and *foaf:lastName* and the value of property *foaf:homepage* of the organization related to ?x, by the composition of the properties *la:refersToWorkedFor* and *la:refersToOrg*

Finally, an example of a *linkset view definition* is $\boldsymbol{l} = (owl{:}sameAs, F_{SWCC}, G_{BrCV}, \pi, \mu)$, where:

- *owl:sameAs* is the property used to indicate that two resources denote the same object;
- F_{SWCC} is the query defined above;
- G_{BrCV} is the query defined above;
- The *alignment* π is the identity permutation;
- The *match predicate* μ is defined as $(s_1, \ldots, s_n, o_1, \ldots, o_n) \in \mu$ iff $\sigma(s_k, o_k) \geq \alpha$, for each $k=1,\ldots,n$, where similarity measure σ is the 3-gram distance (Ngomo and Auer, 2011) and the threshold α is set to *0.5*. That is, since each pair (s_k, o_k) to be compared is a pair of strings, the user might decide to use the same string similarity measure and the same threshold, $\alpha = 0.5$, for all $k=1,\ldots,n$.

3.2 Creating sameAs Linksets

A *sameAs* link is a link of the form *(s, owl:sameAs, o)* and indicates that *s* and *o* denote the same object. Given two different datasets *T* and *U*, a user would like to find *sameAs* links where *s* is defined in *T* and *o* is defined in *U*.

PUC-Rio - Certificação Digital Nº 1321832/CA

There are a few ways to create *sameAs* links. A brute-force solution would be to manually find and create *sameAs* links. A better solution is the sameAs linkset materialization, in which the links are automatically created based on the property values of the datasets. Link discovery tools, such as LIMES (Ngomo and Auer, 2011) and Silk (Volz et al., 2009b), help the materialization task, but the user needs to configure the tool and set the link specifications. This configuration can be understood as the specification of a linkset view definition, in which the user needs to know how the datasets *T* and *U* are modeled, so he can create the simple property path queries *F* and *G*.

The configuration step is generally complicated, since the datasets use different ontologies, with heterogeneous vocabularies and distinct strategies to structure the concepts. It also requires that the user understands the semantics of the datasets *T* and *U* in sufficient detail to specify the alignment between the vocabularies of queries *F* and *G*, which in turn defines how the match predicate is applied.

Thus, the benefit of using views lies in that the user who wants to create sameAs links does not need to fully understand the datasets, but only use the views defined over them. The administrator of each dataset should be responsible for publishing one or more view definitions, where:

- Each view definition is simple, which implies that the vocabulary of the view consists of a single class and a list of properties, which should act as an identifier for the instances of the class.
- Each view definition includes a mapping to the underlying dataset, defined by a SPARQL query, transparently to the users.
- Each view is accompanied by metadata that describe the set of instances represented in the view and indicate how its vocabulary is pre-aligned with standard vocabularies.

The user can browse the published metadata to find simple view definitions $v=(V_F, F)$ and $w=(V_G, G)$ and also explore the pre-alignment between the vocabularies V_F and V_G and standard vocabularies. Hence, he/she will only be responsible for the final alignment between the vocabularies.

In the linkset view example of section 3.1.3, the user wanted to find *sameAs* links between researchers in *BrCV* and *SWCC*. For that, he/she created the simple property path queries, F_{SWCC} and $G_{BrCV,}$ and defined the linkset view.

PUC-Rio - Certificação Digital Nº 1321832/CA

Now, suppose that the administrator of each dataset defines views that capture the properties that qualify researchers. Thus, the user would not need to create the queries, but only use the views already defined. For example, a view for *SWCC* could be $ReSWCC = (V_F, F_{SWCC})$, where V_F = {foaf:Person, foaf:firstName, foaf:lastName, foaf:workplaceHomepage}, and for BrCV could be $ReBrCV = (V_G, G_{BrCV})$, where V_G = {foaf:Person, foaf:firstName, foaf:lastName, foaf:workplaceHomepage}. The new linkset view would be in the form $m = (owl:sameAs, ReSWCC, ReBrCV, \pi, \mu)$, which is similar to linkset *l* defined in section 3.1.3, except that the queries are replaced by views.

The key is that the administrators should define the views a priori, and independently of each other, motivated only by the fact that they represent persons with certain specific profiles. Hence, it would be important to pre-align the vocabulary of each view with the FOAF vocabulary, making it trivial to align the vocabularies of both views definitions.

To conclude, this section illustrated how view definitions simplify the process of *sameAs* linkset materialization.

PUC-Rio - Certificação Digital Nº 1321832/CA

4
Incremental Linkset Maintenance

This chapter describes our strategy and implementation for the incremental maintenance of materialized linksets. Section 4.1 discusses the main problems of applying known strategies for view maintenance in the context of linkset views. Section 4.2 shows how to overcome the problems of the incremental strategy and adapt it to our context. Section 4.3 presents the *Linkset Maintainer* tool, along with its architecture and process. Sections 4.4, 4.5, 4.6 and 4.7 describe, in detail, the main steps of the maintenance process.

PUC-Rio - Certificação Digital Nº 1321832/CA

4.1 Introduction

After the sameAs links are created and materialized, another problem emerges: how to keep the links updated. More precisely, given two datasets, *T* and *U*, and a materialized sameAs linkset *L* from *T* to *U*, the problem now lies in how to maintain *L* when updates on *T* or *U* occur. In traditional view maintenance literature, there are a few alternatives that can help solve this problem.

The first alternative would be to create *versions* of *L* as updates on *T* or *U*, considering that *T* and *U* are also versioned. But this option should be discarded, since we cannot assume that *T* and *U* are versioned. In fact, this alternative would lead to a different set of new problems.

A second approach would be to *rematerialize L*, that is, to recompute *L* when updates are applied to *T* or *U*. This would be a costly alternative, since *L* is computed by a (potentially complex) matching process between property values using queries.

To *invalidate* links in *L* that are affected by updates on *T* or *U* would be another alternative. The problem now lies in that *L* does not contain the triples capturing the property values that generated the sameAs links to detect when an update on *T* or *U* invalidates a sameAs link in *L*.

The last alternative would be to *incrementally maintain L*, that is, to update *L* based on the updates on *T* or *U*. The same problem as for link invalidation would occur, since *L* does not have enough information to recompute a sameAs link, after an update on *T* or *U* occurs.

Motivated by this discussion, section 4.2 presents a strategy that overcomes the lack of information problem and implements the incremental maintenance approach. A simpler version of the strategy would also apply to link invalidation.

4.2 Incremental Strategy

In order to incrementally maintain a materialized linkset *L*, it is necessary to capture how updates on the datasets affect the links and compute the changes that will be applied to *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*). Let u^- be the set of triples affected by the deletions of *u* and u^+ be the set of triples affected by the insertions of *u*.

In the first process required by our incremental strategy, we need to capture the changes that affect each view in $\boldsymbol{V}$. Let *F* be a catalogue view and $F \in \boldsymbol{V}$.

1) Compute a set R^- of resources of view *F* that are affected by triples in u^- (see section 4.6.1).
2) Compute a set R^+ of resources of view *F* that are affected by triples in u^+ (details in section 4.6.1).
3) Retrieve the set *P* of (new) property values (see section 4.6.2).
4) Associate R^- and *P* with the update timestamp t_u.

Recall that $F[\sigma_T(t)]$ denotes the set of triples that *F* returns when executed over state $\sigma_T(t)$ of *T*. The second process of our incremental strategy is required when *F* is a materialized view, that is, $F[\sigma_T(t)]$ is explicit stored as part of a dataset. Let $\sigma_{\boldsymbol{F}}(t_0)$ be the materialized set of triples $F[\sigma_T(t_0)]$. We say that t_0 is the timestamp of the last maintenance of $\sigma_{\boldsymbol{F}}(t_0)$.

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

PUC-Rio - Certificação Digital Nº 1321832/CA

$R^{-}[t_0,t_1]$ 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 $t_i < p(t_u) < t_j$.

We incrementally update $\sigma_F(t_0)$ by following two main steps:

1) Delete from $\sigma_F(t_0)$ all triples whose subject occurs in $R^{-}[t_0,t_1]$.
2) Insert $P[t_0,t_1]$ into $\sigma_F(t_0)$ and obtain $\sigma_F(t_1)$.

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 third process of the incremental strategy, we incrementally update L by 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.

4.3 The Linkset Maintainer Tool

The *Linkset Maintainer* tool was developed to test the strategy outlined in section 4.2. The implementation used the Java 7 programming language, the Eclipse Luna IDE, JBoss Application Server 7 and Jena ARQ API (as the SPARQL Processor).

4.3.1. Architecture

Figure **7** summarizes the architecture of the tool considering the scenarios where the linkset is defined over a virtual view or when it is defined over a materialized view. The *Master Controller* for a view F over a dataset T has the following functionality:

- Normalize the view F defined by the administrator (section 4.4).
- Accept registrations from *View Controllers* and *Linkset Controllers* that will consume data through F (section 4.5).
- Monitor each update on T that affects F and compute the sets R^{-}, R^{+} and P (section 4.6).

PUC-Rio - Certificação Digital Nº 1321832/CA

- Send the sets R^- and P to the *View Controllers* and *Linkset Controllers* registered with itself; the *Master Controller* sends the sets in batch, starting from a given timestamp.

The *View Controller* for a materialized view $\sigma_F(t_0)$, defined over view F, has the following functionality:

- Register itself with the *Master Controller* for F *and* initialize $\sigma_F(t_0)$.
- Request the sets of R^- and P to the *Master Controller* for F and update $\sigma_F(t_0)$ accordingly; the *View Controller* receives the sets in batch, starting from a given timestamp.

The *Linkset Controller* for a linkset L, defined over views F and G, has the following functionality:

- Register itself with the *Master Controllers* for F *and* G and initialize L.
- Request the sets of R^- and P to the *Master Controller* for F and G and update L accordingly; the *Linkset Controller* receives the sets in batch, starting from a given timestamp.

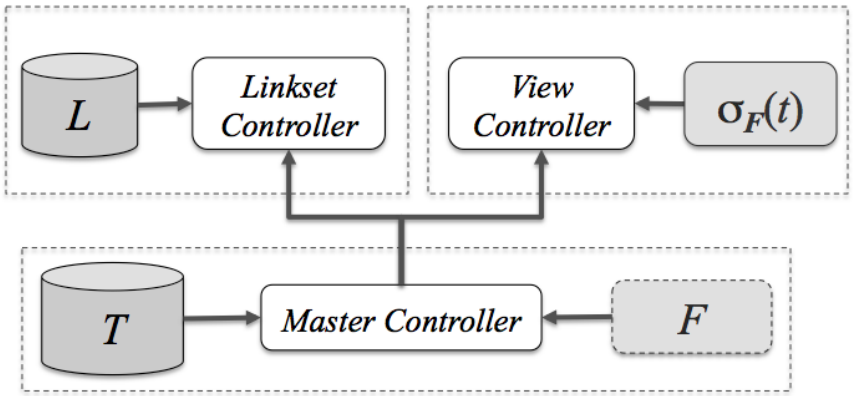

Figure 7 - Linkset Maintainer Architecture

4.3.2. Process Overview

Before starting the maintenance process of materialized linksets, two important tasks need to be executed at design time. The first task will be executed right after the administrator defines the views, in which the Master Controller normalizes the defined view. This task will be better explained in section 4.4. The second task is the process of initializing the linkset and, if necessary, a materialized view, which will be described in section 4.5.

After the initialization, the maintenance process takes place at execution time. Figure **8** shows a sequence diagram for the maintenance process. Steps 1 to 6

PUC-Rio - Certificação Digital Nº 1321832/CA

are explained in detail in section 4.6. Section 4.7 explains steps 7 and 8 and section 4.8 explains steps 9 and 10.

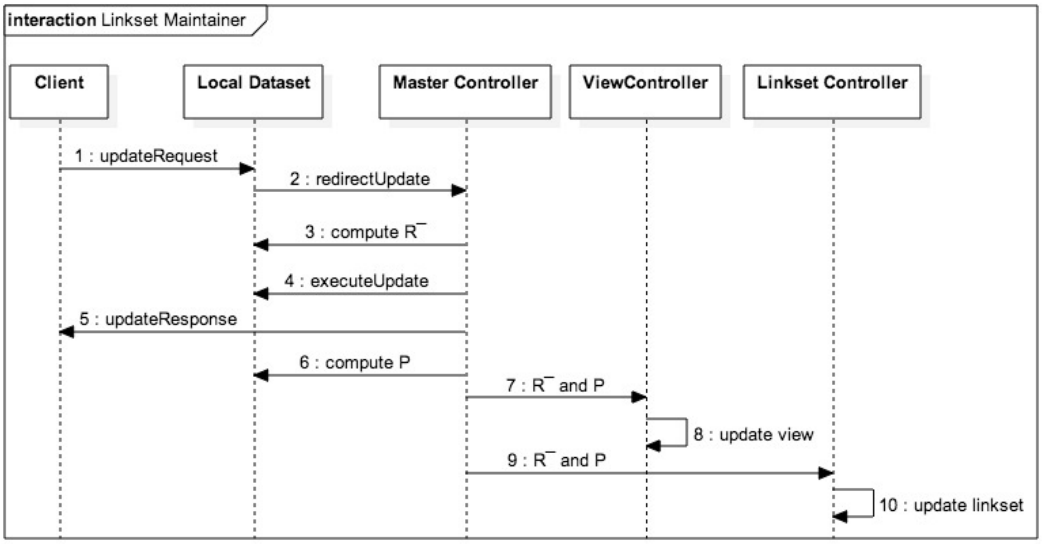

Figure 8 - Sequence Diagram of the Linkset Maintainer

PUC-Rio - Certificação Digital Nº 1321832/CA

4.4 Step 1 – Defining the Views

4.4.1. Overview

The process begins with the administrators defining views for their datasets. As stated in section 3.2, the view definition must be simple. However, the WHERE clause that maps the underlying data is very flexible and can be as much complex as the administrator wants.

For instance, the administrator can use complex *property paths*, as will be described in section 4.4.3. In addition to the complex *property paths*, the administrator is also allowed to use more complex elements than *triples*, as will be described in section 4.4.2.

Monotonicity

The only restriction in the view definition is about the use of *negations*, such as *negated paths, filter not exists* and *minus*, because we need the view to be *monotonic*. Monotonicity permits us to consider only deletions when constructing the set R^- and, likewise, only insertions when constructing R^+.

For example, suppose that the administrator of a dataset T defines the following view F:

```
CONSTRUCT { ?x :p1 ?y }
FROM <T>
WHERE { ?x :p1 ?y .
        FILTER NOT EXISTS { ?x :p2 ?y } }
```

and suppose that *T* has only the following triples:

```
:s1 :p1 :o1
:s2 :p1 :o2
```

Note that a materialization of *F* would also contain only the above triples. Now, suppose that we execute the following update:

```
INSERT DATA { :s1 :p2 :o1 }
```

Note that, if we rematerialize view *F* after the update, the triple ":s2 :p1 :o2" would not be part of the view anymore. Then, an insertion in the dataset actually caused a deletion from the materialized view. Likewise, this same insertion could cause a deletion from a materialized linkset that uses *F*. Therefore, we adopted the monotonicity restriction to simplify the solution.

PUC-Rio - Certificação Digital Nº 1321832/CA

Running Example

Continuing the running example, suppose that the administrator of the fictitious *BrCV* dataset wants to define a view that corresponds to a catalogue of all researchers. He/She has to provide to the Master Controller a JSON (JavaScript Object Notation) file containing the following properties:

- Name: a unique identifier for the view in the JSON file
- Definition: the view itself, that is, the `CONSTRUCT` query.

An example of the JSON file describing a view for researchers from *BrCV* is shown below.

```
{"views":[
     {"name": "Lattes_Researchers",
      "definition":
      "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
       PREFIX foaf: <http://xmlns.com/foaf/0.1/>
       PREFIX la: <http://onto.lattes.br/>
       CONSTRUCT { ?x rdf:type foaf:Person       .
                   ?x foaf:firstName         ?fn .
                   ?x foaf:lastName          ?ln .
                   ?x foaf:workplaceHomepage ?op  }
       FROM <http://lattes.br/sparql>
       WHERE { ?x rdf:type la:Curriculum   .
               ?x foaf:firstName       ?fn .
               ?x foaf:lastName        ?ln .
     ?x la:refersToWorkedFor/la:refersToOrg/foaf:homepage ?op} "
```

```
      }]
}
```

Once the view definition is informed, the Master View Controller has to identify the *predicate triple patterns*, that is, *triple patterns* with predicate paths or predicate variables. In order to help this task, the Master View Controller has to normalize the view, that is, simplify triples by replacing *property paths* for standard triples, where possible.

For instance, the *Sequence Path* of the view "Lattes_Researchers" would be replaced by the following triples:

```
?x la:refersToWorkedFor ?p1 .
?p1 la:refersToOrg ?p2.
?p2 foaf:homepage ?op
```

Then, the list of *predicate triple patterns* of view "Lattes_Researchers" would be:

```
[ ?x rdf:type la:Curriculum,
  ?x foaf:firstName ?fn,
  ?x foaf:lastName ?ln,
  ?x la:refersToWorkedFor ?p1,
  ?p1 la:refersToOrg ?p2,
  ?p2 foaf:homepage ?op ]
```

4.4.2. Normalizing Pattern Elements

The `WHERE` clause of the view is also called the *pattern* of the view. The *pattern* is composed by a group of elements. The most common element is the *triple block*, that is, a continuous list of *triple patterns*. There is also the *filter* element, the *union* element, the *sub query* element, and so on.

For instance, Figure **9** describes the elements of a view of actors and actresses that have less than 70 years old.

PUC-Rio - Certificação Digital Nº 1321832/CA

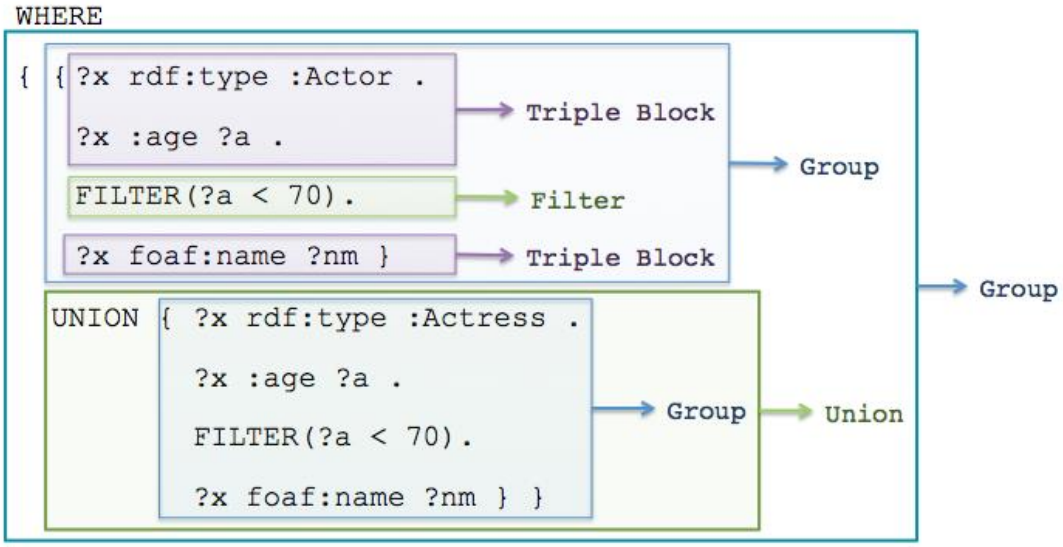

Figure 9 - Elements of a View Pattern

As the *pattern* can be very complex, with several elements inside other elements, the algorithm has to recursively run through these elements and replace each element by its normalized version. When the element is a *triple block*, the Master Controller has to run through all its triple patterns and normalize them. A single *triple block* can be transformed in two blocks, as in the case of *Alternative Paths* described in section 4.4.3. That is why, given one *triple block*, Algorithm 2 returns a list of normalized *triple blocks*, and consequently, Algorithm 1 returns a list of normalized elements. The algorithm for sweeping pattern elements is shown below.

```
Algorithm 1: Sweeper for Pattern Elements
Input: Element
Output: List of Normalized Elements
        List of Predicate Triple Patterns
Method:
     Lnorm = new list of elements
     LPTP = empty list of triple patterns
     add the input to Lnorm
           if the input is a Triple Block
           Lblock = call Algorithm 2 (section 4.4.3) with the input
           Add the second output of Algorithm 2 to LPTP
     return Lblock
           else if the input is a Group
           for each element e in Group g do
                 Lsub = recall this method with e
                 Lnew = new empty list of elements
                 for each element f in Lnorm
                       for each item j in Lsub
                       gnew = replace e for j in f
                       add gnew to Lnew
                 Lnorm = Lnew
```

PUC-Rio - Certificação Digital Nº 1321832/CA

```
                return Lnorm
          else if the input is a Union
          for each element e in Union u do
                Lsub = recall this method with e
                Lnew = new empty list of elements
                for each element f in Lnorm
                      for each item j in Lsub
                      unew = replace e for j in f
                      add unew to Lnew
                Lnorm = Lnew
          return Lnorm
          else if ...
     ...
     return Lnorm
```

Once a normalized list is obtained, all elements are combined in a single query by a UNION clause. Then, the Master Controller updates the JSON file by adding the combined query as the value of "normalized" and the list of predicate triple patterns as the value of "PTP".

In the running example of view "Lattes_Researchers", the JSON file would look like the following:

```
{"views":[
     {"name": "Lattes_Researchers",
      "definition": "…",
      "normalized":
      "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
       PREFIX foaf: <http://xmlns.com/foaf/0.1/>
       PREFIX la: <http://onto.lattes.br/>
       CONSTRUCT { ?x rdf:type foaf:Person       .
                   ?x foaf:firstName         ?fn .
                   ?x foaf:lastName          ?ln .
                   ?x foaf:workplaceHomepage ?op  }
       FROM <http://lattes.br/sparql>
       WHERE { ?x rdf:type la:Curriculum   .
               ?x foaf:firstName       ?fn .
               ?x foaf:lastName        ?ln .
               ?x la:refersToWorkedFor ?p1 .
               ?p1 la:refersToOrg ?p2 .
               ?p2 foaf:homepage ?op }",
      "PTP": "?x rdf:type la:Curriculum,
     ?x foaf:firstName ?fn,
     ?x foaf:lastName ?ln,
     ?x la:refersToWorkedFor ?p1,
     ?p1 la:refersToOrg ?p2,
     ?p2 foaf:homepage ?op "
}
```

PUC-Rio - Certificação Digital Nº 1321832/CA

4.4.3. Normalizing Triple Blocks

In this section we describe an algorithm for *sweeping triple blocks* that iteratively runs through the triple patterns of a block and normalize each triple, replacing complex *property paths* by simpler ones. The iteration stops when all *property paths* are reduced to *Predicate Paths*, that is, paths of length one. Table 2 shows how the normalization is performed when a block has a given *property path*.

Property Path	***Original Block***	***Normalized Blocks***
Inverse Path	`?x ^elt ?y`	(1) `?y elt ?x`
Sequence Path	`?x elt1/elt2 ?y`	(1) `?x elt1 ?o1 .` `?o1 elt2 ?y`
Alternative Path	`?x elt1|elt2 ?y`	(1) `?x elt1 ?y` (2) `?x elt2 ?y`
Fixed Length Path (n > 0)	`?x elt{n} ?y`	(1) `?x` $elt_1/\ldots/elt_n$ `?y`

One or More Path	`?x elt+ ?y`	(1) `?x elt* ?o1 .`
		`?o1 elt ?o2 .`
		`?o2 elt* ?o1 .`

Zero or More Path	`?x elt* ?y`	(1) `?x elt+ ?y` (2) `?x elt{0} ?y`
Zero or One Path	`elt?`	(1) `?x elt ?y` (2) `?x elt{0} ?y`

Table 2 - Property Path Normalization

Note that a *property path* generates one or more simpler *property paths* in a single block, in the case of *Inverse Paths*, *Sequence Paths*, *Fixed Length Path* and *One or More Path* expressions. Also note that a *property path* generates two simpler *property paths* in different blocks, in the case of *Alternative Path*, *Zero or More Path* and *Zero or One Path*. Finally, recall that different *blocks* will be combined into a single query with a `UNION` clause.

The algorithm for *sweeping triple blocks* is shown below. Note that, as the Jena API normalizes *Sequence Paths*, *Inverse Paths* and *Fixed Length Paths*, the algorithm does not handle these *property paths*. The output of the algorithm is a normalized triple block and the list of *predicate triple patterns.*

Algorithm 2: Sweeper for Triple Blocks
Input: Triple Block

PUC-Rio - Certificação Digital Nº 1321832/CA

```
Output: Normalized Triple Block
     List of Predicate Triple Patterns
Method 1:
           b = apply Jena's function in the Triple Block input
           L = add b to a list of Triple Blocks
           LPTP = empty list of triple patterns
           LAUX = new empty list of Triple Blocks
           LVISIT = new empty list of triples that were already
visited
           e = true
           while e do
           for each block b in L do
                 b = apply Jena's function in b
                 add b to LAUX
     for each triple t in b that is not in LVISIT do
           p = the property path of t
           if p is a Predicate Path
           add t to LPTP
           else if p is an Alternative Path
                 LAUX = call Algorithm 3
           else if p is One or More Path
     LAUX = call Algorithm 4
           else if p is an Zero or More Path
     LAUX = call Algorithm 5
           else if p is an Zero or One Path
     LAUX = call Algorithm 6
           if L is not equal to LAUX
                 L = LAUX
                 clear LAUX
           else
                 e = false
           return L and LPTP
```

As an example of the iterations of the algorithm, consider the following *triple block*:

```
 ?x ^foaf:member/foaf:page ?pg
```

In the first iteration, the *Sequence Path* is normalized as follows:

```
?x ^foaf:member ?p1 . ?p1 foaf:page ?pg
```

In the second iteration, the *Inverse Path* is normalized as follows:

```
?p1 foaf:member ?x . ?p1 foaf:page ?pg
```

Then, the list of predicate triple patterns (L_{PTP} in Algorithm 2) for this case simply is:

```
[?p1 foaf:member ?x , ?p1 foaf:page ?pg]
```

The remainder of this section presents details of the normalization of *Alternative Paths*, *One or More Paths*, *Zero or More Paths* and *Zero or One Paths* expressions.

PUC-Rio - Certificação Digital Nº 1321832/CA

Alternative Paths

If the left and right paths of an *Alternative Path* are *Predicate Paths*, we consider that this is a simple *Alternative Path*. A simple *Alternative Path* can be directly identified as a *predicate triple pattern* without the need to be normalized.

For instance, consider the following simple *Alternative Path*:

```
 ?x foaf:name|foaf:givenName ?nm
```

The list of predicate triple patterns in this case is:

```
[?x foaf:name ?nm, ?x foaf:givenName ?nm]
```

However, a complex *Alternative Path* expression has to be transformed into two simpler *property paths* in different groups combined by a UNION clause.

For instance, consider the following complex *Alternative Path*:

```
 ?x (foaf:knows/foaf:name)|(foaf:knows/foaf:givenName) ?nm
```

In the first iteration, this *Alternatve Path* is transformed into the following blocks:

```
(1) ?x foaf:knows/foaf:name ?nm
(2) ?x foaf:knows/foaf:givenName ?nm
```

In the second iteration, the *Sequence Paths* are normalized as the following:

```
(1) ?x foaf:knows ?p1 . ?p1 foaf:name ?nm
(2) ?x foaf:knows ?p2 . ?p2 foaf:givenName ?nm
```

And, the list of predicate triple patterns (L_{PTP}) is:

```
[?x foaf:knows ?p1 , ?p1 foaf:name ?nm,
 ?x foaf:knows ?p2 , ?p2 foaf:givenName ?nm ]
```

The algorithm to process *Alternative Paths* is shown below.

```
Algorithm 3: Alternative Path Normalizer
Input: Triple t with an Alternative Path
     Triple Block b
     List of Triple Blocks L_AUX
     L_PTP - List of Predicate Triple Patterns
Output: List of Normalized Triple Blocks
Method 1:
          p = the property path of t
     t_LEFT = replace p by its left path in t
     t_RIGHT = replace p by its right path in t
     if the predicates of t_LEFT and t_RIGHT are Predicate Paths
     add t_LEFT and t_RIGHT to L_PTP
     else
     b_LEFT = replace t by t_LEFT in b
     b_RIGHT = replace t by t_RIGHT in b
     remove b from L_AUX
```

PUC-Rio - Certificação Digital Nº 1321832/CA

```
     add b_LEFT and b_RIGHT in L_AUX
     return L_AUX
```

One or More Path

A *One or More Path* expression has to be transformed into a combination of *Zero or More Path* expressions with a *One Path* expression.

For instance, consider the following *One or More Path* expression that returns the female ancestors of Alice:

```
?x (:fatherOf|:motherOf)+ ?y .
 ?y foaf:name "Alice"
```

The *One or More Path* expression is transformed into the following *triple block*:

```
 ?x (:fatherOf|:motherOf)* ?p1 .
 ?p1 (:fatherOf|:motherOf) ?p2 .
 ?p2 (:fatherOf|:motherOf)* ?y .
 ?y foaf:name "Alice"
```

Since we need to have at least one path of the form (:fatherOf|:motherOf) between ?x and ?y we can expose an *One Path* expression and place it between two *Zero or More Path* expressions. The *One Path* expression (:fatherOf|:motherOf) can then be treated as a normal *Alternative Path*, which will be normalized in the next iteration. Furthermore, the *Zero or More Path* expressions have to be marked as "visited", to avoid a loop in the algorithm.

The algorithm to process *One or More Path* expressions is shown below.

```
Algorithm 4: One or More Path Normalizer
Input: Triple t with an One or More Path
     Triple Block b
     List of Triple Blocks L_AUX
          List of triples that were already visited L_VISIT
Output: List of Normalized Triple Blocks
Method 1:
          p = the predicate of t
     t_SUB = replace p by its sub path in t
     s = the subject of t
     o = the object of t

     t_ONE = replace o by "?p1" and "+" by "*" in t
     t_TWO = replace s by "?p1" and o by "?p2" in t_SUB
     t_THREE = replace s by "?p2" and "+" by "*" in t
     add t_ONE and t_THREE to L_VISIT

     b_SUB = remove t and add t_ONE , t_TWO and t_THREE to b
     remove b from L_AUX
```

PUC-Rio - Certificação Digital Nº 1321832/CA

```
     add bSUB to LAUX

     return LAUX
```

Zero or More Path

As for a *Zero or More Path* expression, it has to be transformed into a *One or More Path* expression and a *Zero Path* expression.

For instance, consider the following *Zero or More Path* expression that also returns Alice herself, besides her ancestors:

```
  ?x (:fatherOf|:motherOf)* ?y .
  ?y foaf:name "Alice"
```

The *Zero or More Path* expression will be transformed into the following *triple blocks*:

```
(1) ?x (:fatherOf|:motherOf)+ ?y .
    ?y foaf:name "Alice"
(2) ?x (:fatherOf|:motherOf){0} ?y .
    ?y foaf:name "Alice"
```

Note that the Zero or More Path expression will be the union of a *One or More Path* expression and a *Zero Path* expression. In order to save one iteration, we can normalize the *One or More Path* expression directly. Furthermore, the *Zero Path* expression also has to be marked as "visited" since it does not require further normalization.

The algorithm to process *Zero or More Path* expressions is shown below.

```
Algorithm 5: Zero or More Path Normalizer
Input: Triple t with an Zero or More Path
     Triple Block b
     List of Triple Blocks LAUX
     List of triples that were already visited LVISIT
Output: List of Normalized Triple Blocks
Method 1:
          p = the predicate of t
     tSUB = replace p by its sub path in t
     s = the subject of t
     o = the object of t

     //One or More Path
     tONE = replace o by "?p1" in t
     tTWO = replace s by "?p1" and o by "?p2" in tSUB
     tTHREE = replace s by "?p2" in t
     add tONE and tTHREE to LVISIT
     bONE = remove t and add tONE , tTWO and tTHREE to b

     //Zero Path
```

PUC-Rio - Certificação Digital Nº 1321832/CA

```
     t_ZERO = replace "*" by "{0}" in t
     add t_ZERO to L_VISIT
     b_ZERO = replace t by t_ZERO in b

     remove b from L_AUX
     add b_ONE and b_ZERO to L_AUX

     return L_AUX
```

Zero or One Path

A complex *Zero or One Path* has to be transformed into a *Zero Path* expression and a *One Path* expression.

For instance, consider the following *Zero or One Path* expression that returns Alice herself and her father and mother.

```
  ?x (fatherOf|:motherOf)? ?y .
  ?y foaf:name "Alice"
```

The *Zero or One Path* expression will be transformed into the following *triple blocks*:

```
(1) ?x (:fatherOf|:motherOf) ?y .
    ?y foaf:name "Alice"
(2) ?x (:fatherOf|:motherOf){0} ?y .
    ?y foaf:name "Alice"
```

The *One Path* expression (:fatherOf | :motherOf) can also be treated as a normal *Alternative Path* and the *Zero Path* also has to be marked as "visited".

The algorithm to process *Zero or One Path* expressions is shown below.

```
Algorithm 6: Zero or One Path Normalizer
Input: Triple t with an Zero or One Path
     Triple Block b
     List of Triple Blocks L_AUX
Output: List of Normalized Triple Blocks
Method 1:
     L_AUX = list of Triple Blocks input
           p = the predicate of t
     t_SUB = replace p by its sub path in t

     t_ONE = remove "?" from p in t
     t_ZERO = replace "?" by "{0}" in t
     add t_ZERO to L_VISIT

     b_ONE = replace t by t_ONE in b
     b_ZERO = replace t by t_ZERO in b
     remove b from L_AUX
     add b_ONE and b_ZERO to L_AUX

     return L_AUX
```

PUC-Rio - Certificação Digital Nº 1321832/CA

4.5 Step 2 – Initializing Materialized Views and Linksets

Recall the example of section 3.1.3, in which a user would like to create a materialized sameAs linkset between researchers from *BrCV* and *SWCC*. Suppose that the user wants to have the researchers of *BrCV* as a materialized view and the researchers of *SWCC* as a virtual view.

First, in order to materialize the view of researchers from *BrCV*, he/she has to inform to the View Controller the SPARQL endpoint where the view is defined (e.g. "http://lattes.br/sparql"), the name of the view (e.g. "Lattes_Researchers"), and the graph where the view will be materialized (e.g. "http://views/lattes_researchers"). Then, the View Controller will register itself with the Master Controller of view *BrCV* and initialize the materialized view.

Next, the user has to inform the Linkset Controller the SPARQL endpoint where the *source* view is defined (e.g. "http://lattes.br/sparql"), the name of the *source* view (e.g. "Lattes_Researchers"), the SPARQL endpoint where the *target* view is defined (e.g. "http://semanticweb.org/sparql"), the name of the *target* view (e.g. "SWCC_Researchers"), and the graph of the materialized linkset (e.g. "http://linkset/lattes_swcc"). The Linkset Controller sends a request with these parameters to the Master Controllers of the source and the target, which in turn registers the linkset.

For instance, the JSON file of view "Lattes_Researchers" is updated by adding the property "registered":

```
{"views":[
     {"name": "Lattes_Researchers",
      "definition": "…",
      "normalized": "…",
      "map": […]
      "registered":[{"graph":"http://views/lattes_researchers",
                    "timestamp":"0"},
                     {"graph":"http://linkset/lattes_swcc",
                    "timestamp":"0"}]
      }]
}
```

Once the linkset is registered with both views, the Linkset Controller can begin the matching process, in order to initialize the linkset. Since view "Lattes_Researchers" is already materialized, the Linkset Controller can access it through graph "http://views/lattes_researchers". However, as view "SWCC_Researchers" is virtual, the Linkset Controller has to briefly materialize

PUC-Rio - Certificação Digital Nº 1321832/CA

it in a temporary file, just for the initialization process. After the linkset is initialized, the Linkset Controller notifies the Master Controllers, which in turn update the timestamps of the registered linksets.

The Linkset Maintainer Tool uses Silk as the link discovery tool, since it provides an API that enables the matching process to be executed programmatically. The user can choose to use a different discovery tool, but he/she has to manage it apart of the Linkset Maintainer.

Using Silk Single Machine API as the link discovery tool, the user also has to provide a Silk link specification file with linking conditions, defining how the entities of the views should be interlinked. For instance, the file used to match the materialized view of researchers from *BrCV* and the virtual view of researchers from *SWCC* is shown below.

```
<Silk>
  <Prefixes>
    …
  </Prefixes>

  <DataSources>
    <DataSource id="Lattes" type="sparqlEndpoint">
     <Param name="endpointURI" value="http://views/sparql"/>
     <Param name="graph"
           value="http://views/lattes_researchers"/>
    </DataSource>
    <DataSource id="SWCC" type="file">
     <Param name="file" value="../tempTarget.ttl"/>
     <Param name="format" value="TTL"/>
    </DataSource>
  </DataSources>

  <Interlinks>
   <Interlink id="Lattes_SWCC">
      <LinkType>owl:sameAs</LinkType>
      <SourceDataset dataSource="Lattes" var="a">
</SourceDataset>
      <TargetDataset dataSource="SWCC" var="b"> </TargetDataset>
      <LinkageRule>
        <Aggregate type="average">
          <Compare weight="1" metric="levenshteinDistance"
threshold="1">
            <TransformInput function="lowerCase">
              <Input path="?a/foaf:firstName"/>
            </TransformInput>
            <TransformInput function="lowerCase">
              <Input path="?b/foaf:firstName"/>
            </TransformInput>
          </Compare>
```

PUC-Rio - Certificação Digital Nº 1321832/CA

```
          <Compare weight="1" metric="levenshteinDistance"
threshold="1">
            <TransformInput function="lowerCase">
              <Input path="?a/foaf:lastName"/>
            </TransformInput>
            <TransformInput function="lowerCase">
              <Input path="?b/foaf:lastName"/>
            </TransformInput>
          </Compare>
          <Compare weight="1" metric="levenshteinDistance"
threshold="1">
            <TransformInput function="lowerCase">
              <Input path="?a/foaf:workplaceHomepage"/>
            </TransformInput>
            <TransformInput function="lowerCase">
              <Input path="?b/foaf:workplaceHomepage"/>
            </TransformInput>
          </Compare>
        </Aggregate>
      </LinkageRule>
      <Filter/>
      <Outputs>
        <Output id="Linkset" type="sparul">
          <Param name="parameter" value="query"/>
          <Param name="uri" value="http://linkset/lattes_swcc"/>
        </Output>
      </Outputs>
    </Interlink>
  </Interlinks>
</Silk>
```

PUC-Rio - Certificação Digital Nº 1321832/CA

4.6 Step 3 – Computing Affected Resources and New Property Values

After the linkset is initialized, update requests that arrive in a dataset have to be redirected to the Master Controller before they are executed. Given an update u and a view F, the View Controller has to compute sets of resources of F that are affected by u (R^- and R^+) and their new property values (P).

4.6.1. Computing R^- and R^+

Let W_F be the WHERE clause of F and g_F be the graph in the FROM clause of F. Assume that F has already been normalized and let L_{PTP} 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_{PTP} are "$a_k\, b_k\, c_k$", for $k=1,\ldots,n$. The following code shows the template that generates the query to compute R^- for F and u.

```
INSERT { GRAPH <R⁻>
           { ?x :view F .
             ?x :timestamp tu } }
WHERE {
  GRAPH <g⁻> { ?s ?p ?o }
  GRAPH <gF> { WF }
  FILTER( ( ?s=a1 && ?p=b1 && ?o=c1 ) ||
          ( ?s=a2 && ?p=b2 && ?o=c2 ) ||
            …
          ( ?s=an && ?p=bn && ?o=cn ) }
```

The idea of the query is to check if some deleted triple in g^- matches with one of the triple patterns in L_{PTP}. Recall that the variable ?x identifies the resource of the catalogue view as defined in section 3.1.2 and that the query is executed in the old state of the dataset, that is, before u is executed. Also note that the results of the query 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 computing R^+ is similarly defined, except that g^- is replaced by g^+, a named graph for the set of inserted triples u^+, R^- is replaced by R^+ and the query has to be executed after u is executed. After R^- and R^+ are computed we can discard the graphs g^- and g^+. Algorithm 6 summarizes the process of computing the affected resources R^- and R^+.

```
Algorithm 6: Affected Resources Computation Algorithm
Input: c – a changeset
            σT(t0) – the old state of T
Output: R⁻, R⁺ – the graphs with the affected resources
{ Intercept u;
  Populate g⁺ and g⁻;
  Compute R⁻;
  Execute u;
  Compute R⁺;
  Discard g⁺ and g⁻;
  Return R⁻ and R⁺;
}
```

Considering the running example, let v be the normalized view "Lattes_Researchers" and u be the following update:

```
WITH <http://lattes.br/sparql>
DELETE { ?s foaf:firstName "Marco" }
INSERT { ?s foaf:firstName "Marco Antonio" }
```

PUC-Rio - Certificação Digital Nº 1321832/CA

```
WHERE  { ?s foaf:lastName "Casanova" }
```

The triples that compose the DELETE and the INSERT clause are called *quads*. Those are the triples that will actually be removed or added, based on the *query pattern* of the update. For instance, the delete quad of *u* is "?s foaf:firstName "Marco"".

In the first step, we have to populate the graphs g^- and g^+. In the running example, suppose that g^- has "http://lattes.br/deletions" as its URI and g^+ has "http://lattes.br/insertions" as its URI. Then, the Master Controller executes executing the following SPARQL query to populate the graphs:

```
INSERT { GRAPH <http://lattes.br/deletions>
           { ?s foaf:firstName "Marco" } }
INSERT { GRAPH <http://lattes.br/insertions>
           { ?s foaf:firstName "Marco Antonio" } }
WHERE  { GRAPH <http://lattes.br/sparql>
           { ?s foaf:lastName "Casanova" } }
```

Suppose that *BrCV* "http://lattes.br/sparql" contains the following triples:

```
 :Casanova rdf:type la:Curriculum .
 :Casanova foaf:firstName "Marco" .
 :Casanova foaf:lastName "Casanova" .
 :Casanova la:refersToWorkedFor :Casanova_PUCRIO .
 :Casanova_PUCRIO la:refersToOrg :PUCRIO .
 :PUCRIO foaf:homepage "www.puc-rio.br/"
```

Then, graph "http://lattes.br/deletions" contains the following triple:

```
 :Casanova foaf:firstName "Marco"
```

And graph "http://lattes.br/insertions" contains the following triple:

```
:Casanova foaf:firstName "Marco Antonio"
```

In the second step, the Master Controller has to compute the set R^-, in this case, the set of resources of view "Lattes_Researches" that are affected by the deleted triples of *u*. Note that, as this query has fixed template for each view, it can be pre-computed at design time. At execution time, we just need to update the timestamp. Suppose that R^- has "http://lattes.br/deletedResources" as its URI and that the timestamp of the update is "2".

```
INSERT { GRAPH <http://lattes.br/deletedResources>
           { ?x :view "Lattes_Researchers" .
             ?x :timestamp "2" } }
WHERE {
     GRAPH <http://lattes.br/deletions> { ?s ?p ?o }
     GRAPH <http://lattes.br/sparql>
           { ?x rdf:type la:Curriculum   .
             ?x foaf:firstName       ?fn .
```

PUC-Rio - Certificação Digital Nº 1321832/CA

```
            ?x foaf:lastName        ?ln .
            ?x la:refersToWorkedFor ?p1 .
            ?p1 la:refersToOrg ?p2 .
            ?p2 foaf:homepage ?op }
    FILTER(
       (?s = ?x && ?p = rdf:type && ?o = la:Curriculum ) ||
       (?s = ?x && ?p = foaf:firstName && ?o = ?fn ) ||
       (?s = ?x && ?p = foaf:lastName && ?o = ?ln ) ||
       (?s = ?x && ?p = la:refersToWorkedFor && ?o = ?p1 ) ||
       (?s = ?x && ?p1 = la:refersToOrg && ?o = ?p2 ) ||
       (?s = ?x && ?p2 = foaf:homepage && ?o = ?op ) }
```

Then, graph "http://lattes.br/deletedResources" contains the following triples:

```
:Casanova :view "Lattes_Researchers"
:Casanova :timestamp "2"
```

Finally, after R^- is computed, the Master Controller can actually execute the update in the dataset. Now, *BrCV* "http://lattes.br/sparql" contains the following triples:

```
:Casanova rdf:type la:Curriculum .
:Casanova foaf:firstName "Marco Antonio" .
:Casanova foaf:lastName "Casanova" .
:Casanova la:refersToWorkedFor :Casanova_PUCRIO .
:Casanova_PUCRIO la:refersToOrg :PUCRIO .
:PUCRIO foaf:homepage "www.puc-rio.br/"
```

Then, we proceed to compute R^+ executing the following query, supposing that R^+ has "http://lattes.br/insertedResources" as its URI.

```
INSERT { GRAPH <http://lattes.br/insertedResources>
           { ?x :view "Lattes_Researchers" .
             ?x :timestamp "2" } }
WHERE {
     GRAPH <http://lattes.br/insertions> { ?s ?p ?o }
     GRAPH <http://lattes.br/sparql>
           { ?x rdf:type la:Curriculum   .
             ?x foaf:firstName       ?fn .
             ?x foaf:lastName        ?ln .
             ?x la:refersToWorkedFor ?p1 .
             ?p1 la:refersToOrg ?p2 .
             ?p2 foaf:homepage ?op }
     FILTER(
        (?s = ?x && ?p = rdf:type && ?o = la:Curriculum ) ||
        (?s = ?x && ?p = foaf:firstName && ?o = ?fn ) ||
        (?s = ?x && ?p = foaf:lastName && ?o = ?ln ) ||
        (?s = ?x && ?p = la:refersToWorkedFor && ?o = ?p1 ) ||
        (?s = ?x && ?p1 = la:refersToOrg && ?o = ?p2 ) ||
        (?s = ?x && ?p2 = foaf:homepage && ?o = ?op ) }
```

PUC-Rio - Certificação Digital Nº 1321832/CA

Then, graph “http://lattes.br/insertedResources” also contains the following triples:

```
:Casanova :view "Lattes_Researchers"
:Casanova :timestamp "2"
```

4.6.2. Computing *P*

After computing R^- and R^+ we proceed to compute the set of new property values *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*. We can compute *P* simply by executing a query according to the following template.

```
INSERT { GRAPH <P> {
           C_F .
           ?x :view F .
           ?x :timestamp t_u } }
WHERE
{   { { SELECT DISTINCT ?deleted
        WHERE { GRAPH <R⁻> { ?deleted ?p ?o } }
      UNION
      { SELECT DISTINCT ?inserted
        WHERE { GRAPH <R⁺> { ?inserted ?p ?o } }
    }
    GRAPH <g_F> { W_F }
    FILTER ( ( ?x = ?inserted ) || ( ?x = ?deleted ) )
}
```

Note the query to compute *P* also considers the resources in the deleted set R^-. This is necessary since R^- is actually a superset of the set of resources of *F* affected by deletions, denoted S^-. That is, there might be a resource $r \in R^-$ that is not actually affected by the deletions ($r \notin S^-$).

Recall that $F[\sigma_T(t)]$ denotes the set of triples that *F* returns when executed over state $\sigma_T(t)$ of *T*. Formally, a resource $s \in S^-$ iff

$\exists p \exists o((s,p,o) \in F[\sigma_T(t_0)] \wedge (s,p,o) \notin F[\sigma_T(t_1)])$

And,

$S^- \subseteq R^-$

Then, a resource $p \in P$ iff

$\exists x \exists y(p \in (R^- \cup R^+) \wedge (p,x,y) \in F[\sigma_T(t_1)])$

For example, suppose that the administrator of a dataset *T* defines the following view *F*:

```
CONSTRUCT { ?x :p4 ?y }
```

PUC-Rio - Certificação Digital Nº 1321832/CA

```
FROM <T>
WHERE { ?x (:p1|:p2)/:p3 ?y }
```

And suppose that *T* has only the following triples:

```
:s1 :p1 :o1
:s1 :p2 :o1
:o1 :p3 :o2
```

Note that a materialization of *F* would have only the following triple.

```
:s1 :p4 :o2
```

Now, suppose that we execute the following update:

```
DELETE DATA { :s1 :p2 :o1 }
```

Note that the materialization of *F* would not change, since there are two possible paths to :o1 and only one of them is deleted. However, the query to compute R^{-} returns :s1 as a resource affected by deletions. Therefore, the query to compute *P* also needs to consider the deleted resources, so we reinsert into the materialized view or linkset, those resources that was not suppose to be deleted in the first place.

Returning to the running example, the following query computes *P* for view "Lattes_Researchers". Suppose that *P* has "http://lattes.br/newProperties" as its URI.

```
INSERT { GRAPH <http://lattes.br/newProperties> {
           ?x rdf:type foaf:Person       .
           ?x foaf:firstName         ?fn .
           ?x foaf:lastName          ?ln .
           ?x foaf:workplaceHomepage ?op .
           ?x :view "Lattes_Researchers" .
           ?x :timestamp "2" } }
WHERE
{   { { SELECT DISTINCT ?deleted
        WHERE { GRAPH <http://lattes.br/deletedResources>
                { ?deleted ?p ?o } }
      UNION
      { SELECT DISTINCT ?inserted
        WHERE { GRAPH <http://lattes.br/insertedResources>
                { ?inserted ?p ?o } }
    }
    GRAPH <http://lattes.br/sparql>
          { ?x rdf:type la:Curriculum   .
            ?x foaf:firstName       ?fn .
            ?x foaf:lastName        ?ln .
            ?x la:refersToWorkedFor ?p1 .
            ?p1 la:refersToOrg ?p2 .
            ?p2 foaf:homepage ?op }
    FILTER ( ( ?x = ?inserted ) || ( ?x = ?deleted ) ) }
```

PUC-Rio - Certificação Digital Nº 1321832/CA

Then, “http://lattes.br/newProperties” contains the following triples:

```
:Casanova :view "Lattes_Researchers"
:Casanova :timestamp "2"
:Casanova rdf:type foaf:Person .
:Casanova foaf:firstName "Marco Antonio" .
:Casanova foaf:lastName "Casanova" .
:Casanova foaf:workplaceHomepage "www.puc-rio.br/" .
```

4.7 Step 4 – Updating a Materialized Catalogue View

When there is a materialized catalogue view, the corresponding *View Controller* must update it according to the set of resources that were affected after the last timestamp maintenance.

Let $\sigma_F(t_0)$ be a materialized view. First, we need to delete from $\sigma_F(t_0)$ the sets of accumulated deleted resources $\bar{R}[t_0,t_1]$, where t_0 is the last timestamp maintenance of $\sigma_F(t_0)$ and t_1 is the current timestamp. Second, we need to insert into $\sigma_F(t_0)$ the accumulated property values $P[t_0,t_1]$. The following code shows the template of the necessary queries to update the view.

```
DELETE { GRAPH <σF(t)> { ?s1 ?p1 ?o1 }}
INSERT { GRAPH <σF(t)> { ?s2 ?p2 ?o2 }}
WHERE
{  { GRAPH <R̄> { ?s1 :view F .
                 ?s1 :timestamp ?t
                 FILTER ( ?t > t0 )  }
     GRAPH <σF(t)> { ?s1 ?p1 ?o1 }
   }
   UNION
   { GRAPH <P> { ?s2 ?p2 ?o2 .
                 ?s2 :view F .
                 ?s2 :timestamp ?t
                 FILTER ( ?t > t0 )
                 FILTER ( ?p2 != :view &&
                          ?p2 != :timestamp ) }
   }
}
```

Continuing the running example, recall from section 4.5 that the user materialized view “Lattes_Researchers” in graph “http://views/lattes_researchers” and suppose that its last maintenance timestamp was “1”. Then, the View Controller updates the materialized view executing the following query.

```
DELETE { GRAPH <http://views/lattes_researchers>
           { ?s1 ?p1 ?o1 }}
INSERT { GRAPH <http://views/lattes_researchers>
```

PUC-Rio - Certificação Digital Nº 1321832/CA

```
           { ?s2 ?p2 ?o2 }}
WHERE
{  { GRAPH <http://lattes.br/deletedResources>
           { ?s1 :view "Lattes_Researchers" .
             ?s1 :timestamp ?t
             FILTER ( ?t > "1" )  }
     GRAPH <http://views/lattes_researchers>
           { ?s1 ?p1 ?o1 }
   }
   UNION
   { GRAPH <http://lattes.br/newProperties>
           { ?s2 ?p2 ?o2 .
             ?s2 :view "Lattes_Researchers" .
             ?s2 :timestamp ?t
             FILTER ( ?t > "1" )
             FILTER ( ?p2 != :view &&
                      ?p2 != :timestamp ) }
   }
}
```

PUC-Rio - Certificação Digital Nº 1321832/CA

4.8 Step 5 – Updating a Materialized Linkset

Finally, the *Linkset Controller* can update the materialized linkset 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 following template.

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

Continuing the following example, recall from section 4.5 that the user materialized a linkset of researchers in graph “http://linkset/lattes_swcc” and suppose that its last maintenance timestamp was “1”. Then, the Linkset Controller updates the materialized linkset executing the following query.

```
DELETE{ GRAPH <http://linkset/lattes_swcc>
           { ?s ?p ?o } }
WHERE
{ GRAPH <http://lattes.br/deletedResources>
     { ?s :view "Lattes_Researchers" .
       ?s :timestamp ?t
```

```
       FILTER ( ?t > "1" )  }
  GRAPH <http://linkset/lattes_swcc>
     { ?s ?p ?o }
}
```

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. This can be done by the Linkset Controller using the following specification file of Silk.

```
<Silk>
  <Prefixes>
    …
  </Prefixes>

  <DataSources>
    <DataSource id="Lattes" type="sparqlEndpoint">
     <Param name="endpointURI" value="http://lattes.br/sparql"/>
     <Param name="graph"
           value="http://lattes.br/newProperties"/>
    </DataSource>
    <DataSource id="SWCC" type="file">
     <Param name="file" value="../tempTarget.ttl"/>
     <Param name="format" value="TTL"/>
    </DataSource>
  </DataSources>

  <Interlinks>
   <Interlink id="Lattes_SWCC">
      <LinkType>owl:sameAs</LinkType>
      <SourceDataset dataSource="Lattes" var="a">
          <RestrictTo>
              ?b :view "Lattes_Researchers" .
              ?b :timestamp ?t .
              FILTER ( STR(?t) > "1" )
           </RestrictTo>
      </SourceDataset>
      <TargetDataset dataSource="SWCC" var="b"> </TargetDataset>
      <LinkageRule>
        <Aggregate type="average">
          <Compare weight="1" metric="levenshteinDistance"
threshold="1">
            <TransformInput function="lowerCase">
              <Input path="?a/foaf:firstName"/>
            </TransformInput>
            <TransformInput function="lowerCase">
              <Input path="?b/foaf:firstName"/>
            </TransformInput>
          </Compare>
          <Compare weight="1" metric="levenshteinDistance"
threshold="1">
            <TransformInput function="lowerCase">
```

PUC-Rio - Certificação Digital Nº 1321832/CA

```
              <Input path="?a/foaf:lastName"/>
            </TransformInput>
            <TransformInput function="lowerCase">
              <Input path="?b/foaf:lastName"/>
            </TransformInput>
          </Compare>
          <Compare weight="1" metric="levenshteinDistance"
threshold="1">
            <TransformInput function="lowerCase">
              <Input path="?a/foaf:workplaceHomepage"/>
            </TransformInput>
            <TransformInput function="lowerCase">
              <Input path="?b/foaf:workplaceHomepage"/>
            </TransformInput>
          </Compare>
        </Aggregate>
      </LinkageRule>
      <Filter/>
      <Outputs>
        <Output id="Linkset" type="sparul">
          <Param name="parameter" value="query"/>
          <Param name="uri" value="http://linkset/lattes_swcc"/>
        </Output>
      </Outputs>
    </Interlink>
  </Interlinks>
</Silk>
```

PUC-Rio - Certificação Digital Nº 1321832/CA

5
Evaluation and Results

This chapter presents the experiments performed using the Linkset Maintainer Tool to test its performance when dealing with large amount of data and to compare the results with the basic strategy of full recomputation. Section 5.2 describes experiments to update a materialized view "Lattes_Publications". Section 5.3 describes experiments to update a linkset that uses the views "SWCC_Publications" and "Lattes_Publications". Section 5.4 shows an analysis of the updates observed in a real dataset.

PUC-Rio - Certificação Digital Nº 1321832/CA

5.1 Evaluation Setup

In order to compare the performance of the incremental strategy with the full re-computation of sameAs linksets, we selected two datasets:

(1) *SWCC*[4] dump with 320,965 triples.

(2) The *Lattes* curriculum (*BrCV*) of the professors from PUC-Rio, with 11,480,382 triples.

All experiments were executed in an Intel Core i5 1.7 GHz with 4 GB RAM, running OS X Yosemite 10.10.2.

Table 3 shows two views that were elaborated over these datasets with the correspondent name, number of resources and vocabulary.

View	***Resources***	***Vocabulary***
SWCC_Publications	4,243	foaf:Document, dc:title, dc:date
Lattes_Publications	25,092	foaf:Document, dc:title, dc:date

Table 3 - List of Views

[4] http://data.semanticweb.org/dumps/

5.2 Experiments with a Materialized View

First, consider the following definition for view “Lattes_Publications”:

```
CONSTRUCT { ?x rdf:type foaf:Document .
            ?x dc:title ?tl .
            ?x dc:date ?dt .}
FROM <http://lattes.br/sparql>
WHERE { ?x rdf:type la:EventWork .
        ?x la:workTitle ?tl .
        ?x la:workYear ?dt }
```

In the first step of the evaluation, we materialized the above view and computed the runtimes to update it, using the incremental and the full rematerialization strategies.

The updates were performed using the following template, varying the limit and the type of the update (insertion or deletion):

```
INSERT { GRAPH <http://lattes.br/sparql>
          { ?s rdf:type la:EventWork } }
WHERE { GRAPH <http://lattes.br/sparqlCopy>
          { SELECT DISTINCT ?s
            WHERE { ?s rdf:type la:EventWork .
                    ?s la:workTitle ?tl .
                    ?s la:workYear ?dt }
            LIMIT 1 } }
```

Note that these updates affect the view “Lattes_Publications” and that the limit represents the exact number of the affected resources. The following algorithm describes the steps of the experiment, given a limit:

Algorithm 8: Test Runner for a Materialized View
Input: Limit
Output: Runtimes
Method:

```
     //Incremental
     (1) Execute the delete update and compute R¯ and P
     (2) Update the view using the incremental strategy
     (3) Compute the runtime
     (4) Count the number of resources in the materialized view
     (5) Execute the insert update and compute R¯ and P
     (6) Update the view using the incremental strategy
     (7) Compute the runtime
     (8) Count the number of resources in the materialized view

     //Full
     (9) Execute the delete update
     (10) Rematerialize the view
     (11) Compute the runtime
     (12) Count the number of resources in the materialized view
     (13) Execute the insert update
```

PUC-Rio - Certificação Digital Nº 1321832/CA

```
(14) Rematerialize the view
(15) Compute the runtime
(16) Count the number of resources in the materialized view

//Test accuracy
(17) Compare (4) with (12)
(18) Compare (8) with (16)
```

Note that the delete update is always executed before the insert update. Additionally, in order to test the accuracy of the incremental approach, we compared the states of the view after each update with the states of the view using the full rematerialization approach. The accuracy results were positive and all tests passed.

Figure 10 and Figure 11 shows the results of the runtimes to execute the updates (deletions and insertions), using "Lattes_Publications" as a materialized view and "SWCC_Publications" as a virtual view. For each update, the runtimes of the incremental strategy included: the time to compute R^{-} and P, execute the update, and update the view. Likewise, the runtimes of the full recomputation included the time to execute the update and the time to rematerialize the view.

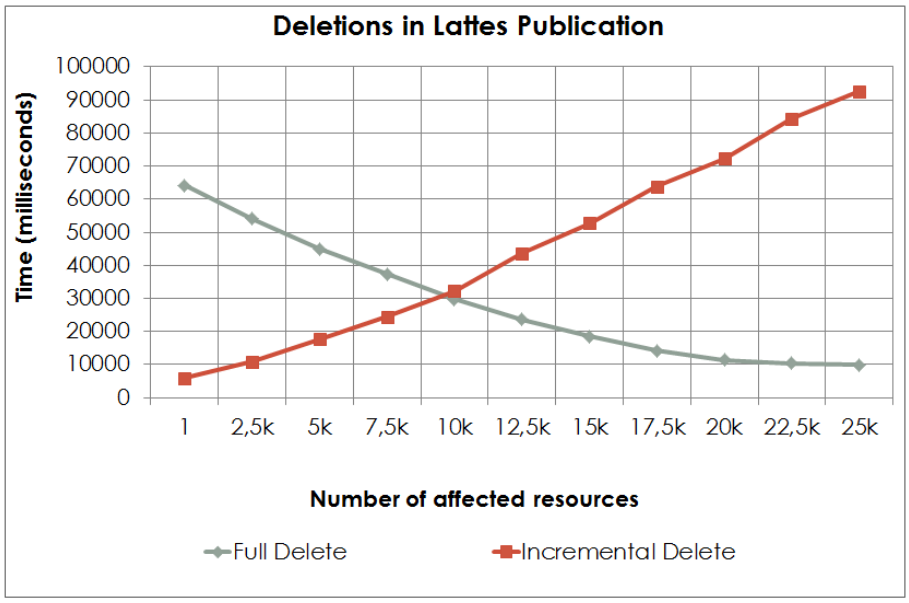

Figure 10 - Deletions on "Lattes_Publications" as a Materialized View

Note that, for the deletions, the full runtime actually drops when the number of affected resources increase. Recall that view "Lattes_Publications" had a total of 25,092 resources; then, the runtime to rematerialize the view is obviously really fast when we delete 25k of these resources. On the other hand, with the incremental strategy, the sets R^{-} and P have 25k resources each, that is, the View Controller deletes 25k resources and tries to reinsert 25k resources (without

PUC-Rio - Certificação Digital Nº 1321832/CA

success), which explains why the runtime of the incremental deletions increases with the number of affected resources.

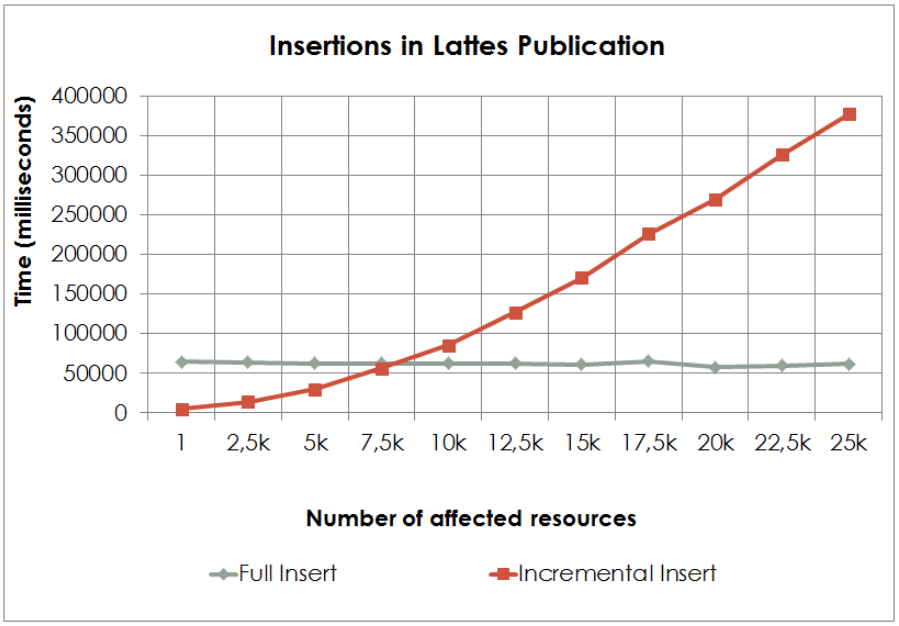

Figure 11 - Insertions on "Lattes_Publications" as a Materialized View

As for the insertions, the full runtime is similar, since we are reinserting the resources that were deleted, which implies that, when view "Lattes_Publications" is rematerialized, it always has 25,092 resources. While in the incremental strategy, if we are inserting 25k resources, the set *P* has 25k resources and the View Controller inserts 25k resources in the materialized view. This explains why the runtime of the incremental strategy is so high when there are a large number of inserted resources.

We stress that the intersection points between the full and the incremental strategies were around 7.5k affected resources for insertions, which is 30% of the total number of resources, and 10k for deletions, which is 40% of the total number of resources. We highlight that the incremental strategy is faster when less than 20% (5k) of total resources of the view are updated, and almost close to zero when few resources are updated.

5.3 Experiments with Linkset Publications

Now consider the following definition for view "SWCC_Publications":

```
CONSTRUCT { ?x rdf:type foaf:Document .
            ?x dc:title ?tl .
            ?x dc:date ?dt .}
```

PUC-Rio - Certificação Digital Nº 1321832/CA

```
FROM <http://data.semanticweb.org/sparql>
WHERE { ?x rdf:type swc:ArgumentativeDocument .
        ?x dc:title ?tl .
        ?x dc:date ?dt }
```

In the second step of the evaluation, we compared the runtime to update a linkset of publications from Lattes and SWCC, using the incremental and the full rematerialization strategies.

This time, we performed updates on SWCC that affected view "SWCC_Publications". The updates used the following template, also varying the limit and the type of the update:

```
INSERT { GRAPH <http://data.semanticweb.org/sparql>
           { ?s rdf:type swc:ArgumentativeDocument } }
WHERE { GRAPH < http://data.semanticweb.org/sparqlCopy>
           { SELECT DISTINCT ?s
             WHERE { ?s rdf:type swc:ArgumentativeDocument .
                     ?s dc:title ?tl .
                     ?s dc:date ?dt }
             LIMIT 1 } }
```

Similarly to Algorithm 8, the following algorithm describes the steps of the linkset experiment, given a limit:

Algorithm 9: Test Runner for Materialized Linkset
Input: Limit
Output: Runtimes
Method:

```
     //Incremental
     (1) Execute the delete update and compute R¯ and P
     (2) Update the linkset using the incremental strategy
     (3) Compute the runtime
     (4) Count the number of links in the linkset
     (5) Execute the insert update and compute R¯ and P
     (6) Update the linkset using the incremental strategy
     (7) Compute the runtime
     (8) Count the number of links in the linkset

     //Full
     (9) Execute the delete update
     (10) Rematerialize the linkset
     (11) Compute the runtime
     (12) Count the number of links in the linkset
     (13) Execute the insert update
     (14) Rematerialize the linkset
     (15) Compute the runtime
     (16) Count the number of links in the linkset

     //Test accuracy
     (17) Compare (4) with (12)
     (18) Compare (8) with (16)
```

PUC-Rio - Certificação Digital Nº 1321832/CA

Figure 12 shows the runtime results to execute the deletions and Figure 13 the insertions, using "Lattes_Publications" as a materialized view and "SWCC_Publications" as a virtual view.

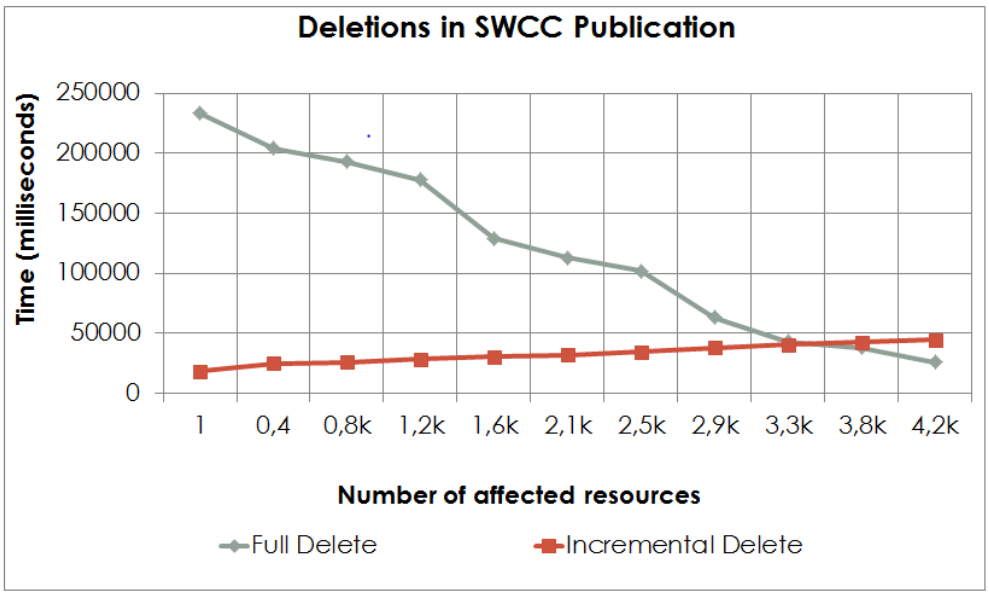

Figure 12 - Deletions on "SWCC_Publications" and Linkset Update

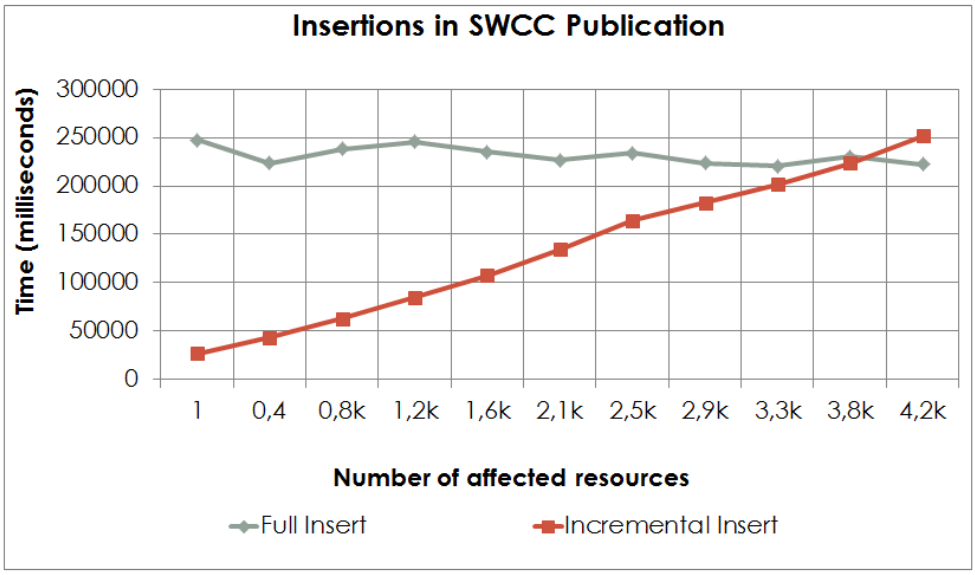

Figure 13 - Insertions on "SWCC_Publications" and Linkset Update

Note that the intersection point between the incremental and the full rematerialization strategies was around 3.3k resources for deletions, and 3.8k for insertions, which are respectively 80% and 90% of the total number of resources of the view (4,243). We highlight that the incremental strategy is four times faster when less than 10% (0.4k) of total resources of the view are updated, and almost close to zero when few resources are updated.

PUC-Rio - Certificação Digital Nº 1321832/CA

5.4 Experiments with the DBpedia Change Sets

In order to show that the incremental strategy is actually effective in practice, in this section, we show that real updates on datasets rarely affect too many resources of a view.

For this, we elaborated three more views about Actresses, Actors and Directors over the DBpedia dataset. Consider the following definition for view "DBpedia_Actress" that resulted in 7,309 resources:

```
PREFIX yago: <http://dbpedia.org/class/yago/>
CONSTRUCT { ?x rdf:type yago:Actor109765278 .
            ?x foaf:name ?nm .
            ?x dbpedia:birthDate ?bd .}
FROM <http://dbpedia.org>
WHERE { ?x rdf:type/rdfs:subClassOf* yago:Actor109765278.
        ?x foaf:name ?nm .
        ?x dbpedia:birthDate ?bd .
        ?x dc:description ?ds .
        FILTER contains(lcase(?ds), "actress") }
```

the following definition for view "DBpedia_Actor" that resulted in 49,308 resources:

```
PREFIX yago: <http://dbpedia.org/class/yago/>
CONSTRUCT { ?x rdf:type yago:Actor109765278 .
            ?x foaf:name ?nm .
            ?x dbpedia:birthDate ?bd .}
FROM <http://dbpedia.org>
WHERE { ?x rdf:type/rdfs:subClassOf* yago:Actor109765278.
        ?x foaf:name ?nm .
        ?x dbpedia:birthDate ?bd .
        ?x dc:description ?ds .
        FILTER contains(lcase(?ds), "actor") }
```

and the following definition for view "DBpedia_Director" that resulted in 9,914 resources:

```
PREFIX yago: <http://dbpedia.org/class/yago/>
CONSTRUCT { ?x rdf:type yago:FilmDirector11008820 .
            ?x foaf:name ?nm .
            ?x dbpedia:birthDate ?bd .}
FROM <http://dbpedia.org>
WHERE { ?x rdf:type/rdfs:subClassOf* yago:FilmDirector110088200.
        ?x foaf:name ?nm .
        ?x dbpedia:birthDate ?bd }
```

In order to show that the number of affected resources is generally small, we analyzed sets from one entire day (April 28, 2015) of DBpedia updates. We calculated the number of updated resources by change set, and we checked how

PUC-Rio - Certificação Digital Nº 1321832/CA

many of these resources were part of our views over DBpedia. Table 4 summarizes the results for the whole day.

	Sum	***Sets***	***Mean***	***Max***
Updated Resources	551,236	5,568	99	975
View Resources	13,199	5,568	2	44

Table 4 - Analysis of DBpedia Change Sets

As we consider that each set is a single update, we have a mean of 99 resources per update, in which only 2% affected some view. Furthermore, we highlight that the max number of resources that affected some view in a single change set was only 44. If we compare with the graphs in section 5.2, the running time is almost close to zero.

PUC-Rio - Certificação Digital Nº 1321832/CA

6 Conclusion

In this work, we presented the Linkset Maintainer, a tool to keep linksets updated, using an incremental strategy. The incremental solution was originally proposed in (Casanova *et. al*, 2014), in which the authors introduced the idea of using views created by the dataset administrators in order to simplify the process of creating materialized linksets. We first showed how to improve and implement the incremental solution that computes only the set of updated resources that are visible through a view. Then, we showed how to keep the views and the linksets updated based on this set.

We also performed some experiments using the proposed approach. The lessons learned were:

- The incremental strategy outperforms full linkset recomputation in most of the cases.
- The incremental strategy outperforms full view recomputation when we update a small number of resources.
- The state of the views and linksets using the incremental strategy is accurate with respect to the state of the full recomputation.
- What most influences the runtime of the incremental strategy is the number of affected resources.
- Typically, the number of affected resources per update is very small.

In order to demonstrate that the incremental strategy outperforms full view and linkset recomputation, we conducted an experiment to measure the performance of both strategies. The results showed that the runtimes of the incremental strategy is almost close to zero when only a few resources are affected. However, when the number of affected resources is close to the total number of resources of the view, the performance of the incremental strategy is worse than that of the full recomputation. Additionally, we showed that the incremental strategy is accurate, comparing the number of materialized links when using both strategies.

PUC-Rio - Certificação Digital Nº 1321832/CA

Finally, we analyzed updates from one day in DBpedia and concluded that, in the experiments, only 2% of the updated resources actually affected some view. We also showed that, in the experiments, the maximal number of view resources for one update was only 44, which means that the running time for the incremental strategy would be much faster than full rematerialization.

We may therefore conclude that the incremental maintenance of materialized sameAs links is efficient in practice since the number of resources that affects a view is typically small. It is worth noting that the strategy can be easily extended to other types of links that can be materialized with link discovery tools. We choose to restrict this dissertation to sameAs links just as a matter of contextualization with the common scenario.

As future work, we plan to continue the development of the tool to improve performance and to provide an interface, so that administrators can define their views and data publisher, and their linksets. Finally, we will make the tool available so anyone can access.

PUC-Rio - Certificação Digital Nº 1321832/CA

7
Bibliography

ALEXANDER, K. et al. **Describing Linked Triplesets with the VoID Vocabulary**. W3C Interest Group Note 03 March, 2011.

BERNERS-LEE, T. **Linked Data**. 2006. Available at: <http://www.w3.org/DesignIssues/LinkedData.html>.

CASANOVA, M. et al. **On Materialized sameAs Linksets**. Database and Expert Systems Applications (pp. 377- 384), 2014.

EUZENAT, J.; SHVAIKO, P. **Ontology Matching**. Springer-Verlag New York, Inc., NJ, USA, 2007.

GUPTA, A.; MUMICK, I.; SUBRAHMANIAN, V. Maintaining Views Incrementally. In Proc. SIGMOD, 157– 166, 1993.

HARRIS, S.; SEABORNE, A. **SPARQL 1.1 Query Language**. W3C Recommendation, 2013. Available at http://www.w3.org/TR/sparql11-query/.

HEATH, T.; BIZER, C. **Linked Data**. Morgan and Claypool Publishers, 2011.

HUNG, E.; DENG, Y.; SUBRAHMANIAN, V. **Maintaining RDF views**. Tech. Rep CS-TR-4612 (UMIACS-TR-2004-54), University of Maryland, 2004.

ISELE, R.; JENTZSCH, A.; BIZER, C. **Efficient Multidimensional Blocking for Link Discovery without losing Recall**. Proc. WebDB, 2011.

MANOLA, F.; MILLER, E. **RDF 1.1 Primer**. W3C Recommendation, 2014. Avaible at <http://www.w3.org/TR/rdf11-primer/>.

MÖLLER, K.; BECHHOFER, S.; HEATH, T. **Semantic Web Conference Ontology**. 2009. Available at: <http://data.semanticweb.org/ns/swc/ontology>.

NGOMO, A.; AUER, S. **LIMES - A Time-Efficient Approach for Large-Scale Link Discovery on the Web of Data**. Proc. IJCAI 2011, pp. 2312–2317, 2011.

NGOMO, A. **On Link Discovery using a Hybrid Approach**. *J. Data Semantics* v.1, pp. 203 – 217, 2012.

POPITSCH, N.; HASLHOFER, B. **DSNotify – A Solution for event detection and link maintenance in dynamic triplesets**. Journal of Web Semantics, 9(3), pp. 266–283, 2011.

STAUDT, M.; JARKE, M. I**ncremental maintenance of externally materialized views**. Proc. VLDB, pp. 75–86, 1996.

PUC-Rio - Certificação Digital Nº 1321832/CA

VIDAL, V.; CASANOVA, M.; CARDOSO, D. **Incremental Maintenance of RDF Views of Relational Data**. Proc. ODBASE 2013, pp 572-587, 2013.

VIDAL, V. *et al*. **Specification and Incremental Maintenance of Linked Data Mashup Views**. Advanced Information Systems Engineering. Springer International Publishing, p. 214-229, 2015.

VOLZ, J.; BIZER, C.; GAEDKE, M. **Web of Data Link Maintenance Protocol** - Maintaining Links Between Changing Linked Data Sources. 2009. Available at: <http://www4.wiwiss.fu-berlin.de/bizer/silk/wodlmp>.

VOLZ, J. et al. **Discovering and Maintaining Links on the Web of Data**. Proc. ISWC 2009, pp. 650-665, 2009.

PUC-Rio - Certificação Digital Nº 1321832/CA