SPARQL Query Rewriting for Implementing Data Integration over Linked Data

Gianluca Correndo,[*] Manuel Salvadores, Ian Millard, Hugh Glaser and Nigel Shadbolt
ECS Department, University of Southampton
SO17 1BJ
Southampton, UK
{gc3, ms8, icm, hg, nrs}@ecs.soton.ac.uk

ABSTRACT

There has been lately an increased activity of publishing structured data in RDF due to the activity of the Linked Data community[1]. The presence on the Web of such a huge information cloud, ranging from academic to geographic to gene related information, poses a great challenge when it comes to reconcile heterogeneous schemas adopted by data publishers. For several years, the Semantic Web community has been developing algorithms for aligning data models (ontologies). Nevertheless, exploiting such ontology alignments for achieving data integration is still an under supported research topic. The semantics of ontology alignments, often defined over a logical frameworks, implies a reasoning step over huge amounts of data, that is often hard to implement and rarely scales on Web dimensions. This paper presents an algorithm for achieving RDF data mediation based on SPARQL query rewriting. The approach is based on the encoding of rewriting rules for RDF patterns that constitute part of the structure of a SPARQL query.

Categories and Subject Descriptors

H.2 [**Database Management**]: Distributed Databases—*query processing, database integration, XML/XSL/RDF*

General Terms

SPARQL, query rewriting, RDF

Keywords

SPARQL, query rewriting, RDF

1. INTRODUCTION

The Semantic Web is aiming to provide the enabling technologies to effectively publish, retrieve and integrate structured data on the Web. In recent years an increasing number of data sets, published in RDF format, is emerging, fuelled primarily by the efforts of the Linked Data community that advocates the adoption of simple design principles[2] in order to create a "Web of Data".

Key factors in the success of such a vision of a network of machine readable information is the establishing of a set of wide used standards and procedures, and can be summarised in four main points:

- Identify resources with URIs.
- Use of HTTP URIs instead of proprietary schemes
- Use resolvable URIs so that users can retrieve information about resources using HTTP lookup.
- Data is not a standalone entity, but it must be linked with other data.

In spite of the high expressive power of the languages used to define ontologies (e.g. RDFS, OWL, and SKOS), the wide range of vocabularies within the data cloud restrains the realisation of a machine-processable Semantic Web. The adoption of ontologies in a Web of Data in fact, can not be easily coordinated since there is no overseeing managing entity. There is consequently an issue related to the mediation of different ontologies used by data publishers, for translating data and queries regardless of ontology boundaries.

Another central issue in the Semantic Web research is the notion of identity. In fact the adoption of URIs ensures to uniquely identify (informative) resources in the web, not the entities the resources refer to [19]. Therefore, the more data sets are published, the more significant is the issue of finding overlapping entities (i.e. different URIs that refer to the same **real** entity) and more pressing will be the need of a service of co-reference resolution [16].

The rest of this paper presents an algorithm that exploits both ontology alignments (with particular interest on data manipulation) and entity co-reference resolution (seen as a particular problem of data manipulation) for rewriting queries over RDF and so achieving integration of heterogeneous data sources. The remaining sections are structured

[*]contact
[1]`http://linkeddata.org`

[2]`http://www.w3.org/DesignIssues/LinkedData`

as follows. Section 2 presents an overview of the issues in implementing a distributed information service and solutions proposed in the field relevant for the work presented here. Section 3 presents the formalism used to describe the ontology alignments and the algorithm used for rewriting queries and finally our conclusions presented in Section 4.

2. STATE OF THE ART

Gio Wiederhold [29] in the 90's introduced the idea of the *mediators based architecture* for dealing with heterogeneities of representations in distributed information systems. In his words a *mediator* is a "software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications". Such mediators are therefore meant to make independent applications from the underneath data resources available to the distributed information system. The concept of *mediator*, introduced by Wiederhold, has been then used by many proposals within distributed information systems area (e.g. KRAFT [24], TSIMMIS [15] etc.) for solving heterogeneity of data representations and providing a homogeneous information space to query.

In the case of the Linked Data cloud, data resources are scattered over the Web, with very little connection at an organisational level. Adopted schemas are not designed to support particular applications and there is a pressing need to encode flexible mediators between application oriented schemas and available data schema, for exploiting the increasing amount of information published via RDF.

Defining explicitly data set schemas via ontologies allows for the alignment of different vocabularies and, using such alignments, encode flexible query mediators [14] that can be used in order to integrate data from heterogeneous sources. The Semantic Web community has devoted a particular effort on defining standards for publishing data and ontologies on the web, using it as a globally accessible media for reusing structured data.

Ontology alignment has been studied extensively in literature, trying to discover automatic procedures for discovering such alignments [14] and proposing logical formulations for them [11]. Automatic tools have been provided over the years for aligning ontologies by exploiting information such as lexical similarity, schema similarity, formal concept analysis, and instance learning. Despite the great attention devoted to automate the task of finding ontology alignments, still little effort has been devoted to study how to exploit them for easing the data integration task on a web scale.

Query rewriting [5], in database theory, is one approach to implement view-based query processing, the task of answering a query on a database, exploiting the information on a set of views. Within the Semantic Web community, approaches to query rewriting has been provided [17] using description logics, that augment the usual descriptive power of Datalog with currently adopted standards such as OWL. SPARQL is the W3C recommendation for querying RDF data sets [25], therefore the approach defined in this paper will employs RDF as a data description language and SPARQL syntax and semantics in order to demonstrate the query rewriting capabilities.

Within the Semantic Web community, query rewriting is gaining attention for fulfilling a number of tasks such as: query optimization [28]; query decomposition [2, 26]; query translation [3]; description logic inference [21] and also data integration [23, 1]. Yixin et al. [21] proposed query rewriting in order to implement description logic reasoning in SPARQL query answering. The problem of implementing reasoning over the Semantic Web is that the size of inference models could be prohibitive for web scale deployment. Since many alignment frameworks (e.g. C-OWL [4], OMA [10], and others [12, 22]) employ logic primitives as a language for expressing alignments between ontologies, the very same problem of scalability applies to data integration tasks as well.

Euzenat et al. [13] proposed to use SPARQL query language in order to solve data translation problems relying on its features for extracting data and creating new triples using the CONSTRUCT statement. However, the problem of how to create dynamically such queries, exploiting the alignments that has been declared between ontologies, is still an open issue and a relatively understudied field of research within the Semantic Web community.

Some proposals have been provided recently [23, 7] for adopting query reformulation to produce flexible mediators for OWL. RDF pattern rewriting [21] looks like a good candidate for providing a sound mediating framework based on syntax instead of logical reasoning, more importantly considering that RDF is the data description model adopted as a standard for the Semantic Web.

To date, there are still no consolidated proposal that pragmatically addresses query mediation task in the Semantic Web context where there are still few consolidated standards (i.e. RDF as data model, HTTP for data transport and URI for resource identification and retrieval). Further, the information space on the Web of Data is highly redundant and data repositories need to be integrated in order to provide high recall result sets. The use of semantically reach schemas in form of ontologies poses even further problems when data is to be integrated. Reasoning task, even on simple logic frameworks, does not scale well and data repositories cannot be integrated relying on reasoning on an overall mediating ontology. The approach we are fostering in this paper is about achieving data integration on a data repository basis relying on syntactical transformation of RDF graphs, the common denominator of most ontology languages in the Semantic Web.

The proposal described in this paper takes some of the already existing approaches in the field, in detail [21, 23], about rewriting the RDF graph pattern of a SPARQL query in order to fit a query to a new model. We have augmented such approaches in order to handle data manipulation functions that are usually not taken into account in ontology based data integration techniques whose focus is mainly to reconcile heterogeneous domains conceptualizations. In this work, data manipulation functions are used in order to solve an important problem that would prevent an effective data integration, the problem of entity co-reference. Moreover, this approach employs a declarative formalisation of the alignments between RDF structures that allows to abstract from a particular implementation framework (i.e. rule based, logic programming or ad-hoc implementations). Finally, an approach based on RDF level transformations allows to work with different ontology languages based on RDF (e.g. RDFS, OWL, SKOS etc.)

3. SPARQL QUERY REWRITING

Unlike traditional approaches to data integration where all different schemas are mediated by a global schema used for expressing the queries, here it is possible to express rules for translating queries from schema to schema and also from data set to data set, as it is needed. The approach to data integration is similar to the one adopted in peer data management systems [18] where queries can be rewritten multiple times, depending on where the query will be executed. In our approach therefore we identify the **query**, the **source** ontology used to formulate the query (or data set the query was formulated for) and the **target** ontology (or data set) as the main inputs for implementing an on-the-fly data integration. The **rewritten query** that fits the **target** ontology or data set is the output of the procedure.

Before we start describing the process of query translation in detail, the reader could benefit from knowing the adopted semantics for the RDF triples, in order to make clearer also the following definitions of ontology and entity alignments, triple matching and pattern rewriting. The semantics associated to RDF triples used in this paper is a version of existential binary relational logic, the same adopted in the W3C document describing RDF semantics [20].

Such logic can be obtained by encoding a typical RDF triple $\langle s, p, o \rangle$ as an atomic formula $p(s, o)$, or using a three-place notation $Triple(s, p, o)$ (three-place notation will be used henceforth). The predicate $Triple$ is defined over the domain $I \times I \times (I \cup L)$ where I is the set of all the IRIs[3] and L is the set of all the RDF Literals. Blank nodes are translated, in accordance with RDF semantics specification, as existentially quantified variables, therefore a triple like the following $\langle ex\!:\!a; rdf\!:\!type; \_\!:\!x \rangle$ is translated into the following formula $\exists x(Triple(ex\!:\!a, rdf\!:\!type, x))$. An RDF graph is therefore translated into a logical theory whose formulas are derived by applying the above described semantics to the graph's triples.

3.1 SPARQL Query Anatomy

A SPARQL SELECT query is one form of SPARQL query that returns a selection of all the data in a data set, and it is divided mainly in three main parts:

- Query Result Form: is composed of a list of variables to be returned by the query.
- Basic Graph Pattern (or BGP): is a block of triple patterns that match altogether with the queried graph's triples.
- Filter Section: is an optional part of the BGP above defined, it filters the set of solutions matched by the previous BGP section filtering values bounded to variables.

```
PREFIX id:<http://southampton.rkbexplorer.com/id/>
PREFIX akt:<http://www.aktors.org/ontology/portal#>
SELECT DISTINCT ?a WHERE {
  ?paper akt:has-author id:person-02686 .
  ?paper akt:has-author ?a .
  FILTER (!(?a = id:person-02686 ))
}
```

Figure 1: SPARQL query example

As an example let us consider the SPARQL query given in Figure 1 defined for one of the data repositories maintained by the ReSIST project[4] and published as Linked Data. The data repositories managed by the ReSIST project contain information about academic publications, authors, and research projects in the multidisciplinary domains of Dependability, Security, and Human Factors. The data repositories can contain redundant data, therefore it is important to query all the available repositories in order to increase the recall of the information retrieval task. The SPARQL query depicted in Figure 1 extracts from the *Southampton* data repository all the distinct URIs of the co-authors of a given person identified by the URI `http://southampton.rkbexplorer.com/id/person-02686` (excluding the given URI).

We can identify in such a query an accessory prologue for defining the name-spaces used within the query (keyword `PREFIX`) and then:

- Query Result Form: `SELECT DISTINCT ?a` we ask for all the distinct bindings over the variable `?a`
- Basic Graph Pattern (or BGP): contained within `WHERE {...}` section describe the patterns the resulting triples should all match.
- Filter Section: `!(?a = id:person-02686 )` we ask that the bounded value is different from the one provided (i.e we want all the co-authors of `id:person-02686` **except** `id:person-02686` itself.

A specification for SPARQL language has been provided in literature for relational algebra [8], SQL [6], and for Datalog [27]. For the sake of the presentation of the algorithm, we will consider just the basic graph pattern (or **BGP**) section and we will show how it is rewritten in order to fit a target ontology or data set, different from the source ontology or data set intended for the original query.

3.2 Alignment Model

In this section the model adopted for describing alignments between ontologies (i.e. **O**ntology **A**lignments, **OA** henceforth, Section 3.2.1) and ontology entities (i.e. **E**ntity **A**lignment, **EA** henceforth, Section 3.2.2) will be presented.

[3]IRI stands for International Resource Identifier, a standard from W3C [9]

[4]`http://www.resist-noe.org/`

3.2.1 *Ontology Alignment Definition*

The ontology alignment formalism adopted here addresses the issue of integrating heterogeneous data sets present in the Linked Data cloud. Data sets, in fact, can reuse vocabularies that are more or less standard (e.g. FOAF, Dublin Core, SIOC) and, in doing so, they can introduce a bias in interpreting their semantics. This means that an alignment can be used for a particular data set and not be suitable for another and ontology alignments must be able to represent this kind of scenarios. Consequently, the ontology alignment formalism adopted allows to specify **source** and **target** coordinates for describing the validity of a given alignment. Such coordinates can be as specific as defining a source and a target data set (in this case the entity alignments in EA cannot be reused for another dataset) or as general as defining a source and a target ontology (in this case the entity alignments in EA can be reused for aligning different data sets that adopt such ontologies.

An ontology alignment OA is therefore defined as a quad $OA = \langle SO, TO, TD, EA\rangle$ where:

- $SO \subseteq I$ is the set of URIs of the **S**ource **O**ntologies,
- $TO \subseteq I$ is the set of URI of the **T**arget **O**ntologies,
- $TD \subseteq I$ is the set of URI of the **T**arget **D**ata sets and
- EA is a set of **E**ntity **A**lignments (see Section 3.2.2) relative to OA.

Note that I is the set of all IRIs as defined in Section 3. SO, TO, and TD are used in order to define the context of validity of the entity alignments contained in EA. Representing target ontologies and datasets for a given ontology alignment it is possible to represent alignments that are specific for two given datasets (in this case SO and TD will be used) or general for two given ontologies (in this case SO and TO will be used).

In aligning two data sets it is possible to need more than one ontology alignment since data sets can use different ontologies to define the data. Querying the alignment server we can retrieve all the relevant ontology alignments for integrating two given data sets. The union of the entity alignments belonging to the relevant ontology alignments can then be used in order to rewrite queries between the data sets.

As an example, referring to the data cloud managed by the ReSIST project mentioned in Section 3.1, there is a data set named `KISTI`[5] that adopts a different ontology for describing its data and there is therefore the need to integrate it with the other data sets. The currently adopted ontology within the ReSIST repositories (named RKB) is the `AKT`[6] ontology, while the `KISTI` has developed its own ontology. The ontology alignment will need therefore to mention the `AKT` as the **source** ontology, and the `KISTI` as the **target** ontology and data set. Moreover, the alignment expressed is meant to be local to the target data set (i.e. the `KISTI` RKB repository that adopts the `KISTI` ontology).

[5]Korea Institute of Science and Technology Information: `http://www.kisti.re.kr`

[6]Advanced Knowledge Technologies: `http://www.aktors.org/ontology/support`

The ontology alignment will be as follow:

- $SO =$ `{<http://www.aktors.org/ontology/portal#>}`
- $TO =$ `{<http://www.kisti.re.kr/isrl/ResearchRefOntology#>}`
- $TD =$ `{<http://kisti.rkbexplorer.com/id/void>}`

This ontology alignment definition can then be used to target the given data set (i.e. TD section) and the entities aligned belong to the `AKT` ontology (source ontology, SO section) and to the `KISTI` ontology (target ontology, TO section).

3.2.2 *Entity Alignment Definition*

An entity alignment EA codifies how to rewrite a triple for fitting a new ontology, it defines therefore a pattern rewriting and eventually a set of constraints over variables present in the alignment itself. The alignments so defined are directional (i.e. not symmetric).

An entity alignment EA is defined as a triple $EA = \langle LHS, RHS, FD\rangle$ where:

- LHS (namely **L**eft **H**and-**S**ide): a (usually open) atomic formula using the predicate $Triple$ that contains no functional symbols.
- RHS (namely **R**ight-**H**and **S**ide): a (usually open) conjunctive formula whose atoms are predicates $Triple$ that contain no functional symbols.
- FD (namely **F**unctional **D**ependencies): a set of functional dependencies in the form of
  $var = function(t_1, ..., t_n)$
  where $t_1, ..., t_n$ are all ground terms or variables present in LHS, var is a variable present in the RHS, and $function$ is a symbol that identifies a function known by the system that operate the translation. Note that this function is not necessarily known by the system that will run the resulting query. The functional dependencies are equivalence constraints over variables that must hold in the rewriting process.

The definition of an entity alignment mimics the structure of a rule in a rule-based system (e.g. CLIPS[7] or RuleML[8]) or a Horn clause in logic programming. Although in this work we intended to implement a particular semantics that could be used to translate the intensional definition of an answer set (i.e. a query) and not only grounded data. An entity alignment can therefore be interpreted as a definite Horn clause in first-order logic where only the $Triple$ predicate is used (in accordance with the RDF semantics provided in [20]). Following this interpretation the LHS formula is the

[7]`http://clipsrules.sourceforge.net/`

[8]`http://ruleml.org`

head, the RHS is the body, and the functional dependencies FD encode further equality constraints upon variables that appear either in LHS or RHS.

As an example, considering the previous SPARQL query in Figure 1 and the relative data integration scenario, there is a non trivial matching of the `akt:has-author` object property from the `AKT` ontology in the `KISTI` ontology. The object property `akt:has-author` in fact relates a paper instance to its authors, while in the `kisti` ontology a paper is related to an instance of `kisti:CreatorInfo` and only then it is possible to retrieve the instance of the author via the `kisti:hasCreator` object property.

Using the presented alignment formalism for ontology entities, we can express such a complex alignment in the following way (note that, adopting the RDF syntax, blank nodes are represented as `_:node` and are treated as existentially quantified variables):

- *LHS*: `<_:p1, akt:has-author, _:a1>`
- *RHS*: `{<_:p2, kisti:CreatorInfo, _:c>,`
  `<_:c, kisti:hasCreator, _:a2>}`
- *FD*: `{`
  `_:a2=sameas(_:a1,`
  `"http://kisti.rkbexplorer.com/id/\S*"),`

  `_:p2=sameas(_:p1,`
  `"http://kisti.rkbexplorer.com/id/\S*")`
  `}`

Note that `"http://kisti.rkbexplorer.com/id/\S*"` is not just a string, but a regular expression pattern, in detail the one that accepts all the strings that start with `"http://kisti.rkbexplorer.com/id/"`. The function `sameas` used in the above alignment returns the URI that identifies the same resource as the one provided in input and that satisfies the regular expression pattern provided as second input.

The alignments can then be represented with a language of choice. The representation language adopted in the work herein presented is RDF that is hence the object of our rewriting process as well as a tool for achieving it. An RDF code chunk that encodes the above mentioned alignment between entities can be given, using the more concise Turtle representation syntax[9], as follows:

```
1. @prefix rdf: <http://www.w3.org/1999/02/...>.
2. @prefix map: <http://ecs.soton.ac.uk/om.owl#>.
3. @prefix akt2kisti: <http://ecs.soton.ac.uk/...>.
4. @prefix akt:<http://www.aktors.org/ontology/...>.
5. @prefix kisti:<http://www.kisti.re.kr/isrl/...>.
6. akt2kisti:creator_info
7.  a map:EntityAlignment;
8.  map:lhs  [
9.    rdf:type  rdf:Statement;
10.     rdf:subject _:p1;
11.     rdf:predicate akt:has-author;
12.     rdf:object  _:a1
13. ].
14. map:rhs [
15.   rdf:type  rdf:Statement;
16.     rdf:subject _:p2;
17.     rdf:predicate kisti:CreatorInfo;
18.     rdf:object  _:c
19. ]
20. map:rhs [
21.   rdf:type  rdf:Statement;
22.     rdf:subject _:c;
23.     rdf:predicate kisti:hasCreator;
24.     rdf:object  _:a2
25. ]
26. map:hasFunctionalDependency [
27.   rdf:type  rdf:Statement;
28.     rdf:subject _:a2;
29.     rdf:predicate map:sameas;
30.     rdf:object  (
31.       _:a1 ,
32.       "http://kisti.rkbexplorer.com/id/\S*"
33.     )
34.   ]
35. ]
36. map:hasFunctionalDependency [
37.   rdf:type  rdf:Statement;
38.     rdf:subject _:p2;
39.     rdf:predicate map:sameas;
40.     rdf:object  (
41.       _:p1,
42.       "http://kisti.rkbexplorer.com/id/\S*"
43.     )
44.   ]
45. ]
⋮
```

[9]http://www.w3.org/TeamSubmission/turtle/

A detail, that could be confusing to the reader, is the frequent use of the *reification* mechanism in order to describe and deal with RDF statements (i.e. triples) using triples. In short, since an RDF statement has no associated URI, it should be impossible to describe or say anything at all about that statement. Therefore, for doing so, instead of using the triple itself, an instance of the $rdf : Statement$, which can have an URI, is used. So, instead of using $\langle s, p, p \rangle$ a URI for an entity st can be created and can be used as follows: $\langle st, rdf\!:\!type, rdf\!:\!Statement \rangle$, $\langle st, rdf\!:\!subject, s \rangle$, $\langle st, rdf\!:\!predicate, p \rangle$, $\langle st, rdf\!:\!object, o \rangle$.

Another point to note is the use of URIs for identifying functions in the alignments definition (as you can see from the code above, lines 29 and 39). The adoption of name spaces allows the unique identification of functions across organizations, helping therefore in managing the procedural knowledge needed to manipulate data and exchanging information. The representation of function applications as triples is consistent with the interpretation of functions as relations that bind the input and output values. If we think RDF properties as algebraic relations, then single RDF triples are just statements that enumerate the elements of the relation. Using reification we can express intensionally the relation between two classes of values: the values that variable $?a_1$ can take and the values that the variable $?a_2$ can take. Such declarative knowledge is useful to represent functional dependencies between variables of an entity alignment that can then be used for solving value bindings within RDF graph patterns as described in Section 3.3.

The alignment formalism here described does not rely on description logic semantics that usually supports ontology languages, but it addresses transformations between RDF structures (i.e. graphs), being able therefore to express alignments for different ontology languages based on RDF (i.e. RDFS, OWL, SKOS, etc.). For now the possibility to express RDF graphs transformations is limited, the matching side (the LHS part) in fact can contain only a simple triple (called *statement* in RDF). This simple representation of syntactic transformation of RDF graph patterns allows us to describe alignments of different levels of complexity. Elaborating on the source of [11], and on its definitions, we provided an account of the level of complexity that this formalism is able to express.

Level 0 is the basic definition of correspondence between two discrete entities, identified by a URI. The correspondence is usually an equivalence relation "≡" and it is not dependant on a particular language. Such alignments are easily represented:

Class alignment: between C_1 and C_2
$\forall x(Triple(x, rdf: type, C_1) \rightarrow Triple(x, rdf: type, C_2))$

Property alignment: between P_1 and P_2
$\forall x \forall y(Triple(x, P_1, y) \rightarrow Triple(x, P_2, y))$

Level 1 alignments are still language independent and allow to replace pairs of entities by pairs of sets (or lists) of entities. These kind of alignments are not fully representable by this formalism without requiring the support of RDFS or OWL primitives since the level definition is independent on the language used but not independent on its underlying semantics. This means for example that the `owl:unionOf` primitive requires modification of not only the pattern of RDF triples, but the whole BGP structure, requiring surrogates from SPARQL language (i.e. `UNION`). An example of level 1 alignment that is representable by the presented formalism is the following, aligning an entity to an intersection of entities:

$\forall x(Triple(x, rdf\!:\!type, wine_1\!:\!Burgundy) \rightarrow$
$Triple(x, rdf\!:\!type, wine_2\!:\!Wine) \wedge$
$Triple(x, rdf\!:\!type, goods\!:\!BurgundyRegionProduct))$

It is noteworthy the fact that, in order to exploit alignments that use description logic primitives, such as **subclass**, **disjoint** or **compatible** [4], one would need to do a reasoning step over the data for retrieving all the correct instances.

Level 2 considers sets of expressions of a particular language L where a typical formula $\forall \overline{x_f}(f \Rightarrow \exists \overline{y_g} g)$ (where f and g are formulas of L, $\overline{x_f}$ is the set of variables present in f, and $\overline{y_g}$ is the set of variables present in g) encodes a directional alignment. In such alignments the variables in the head of the formula are universally quantified and the variables in the body of the formula are existentially quantified. Such alignments are partially represented by this formalism when the expressions used are representable using the presented first-order logic with the $Triple$ predicate. An example of level 2 alignment representable with RDF patterns can be given as follows, when a concept in one ontology O_1 is translated with a value partition over a property in a second ontology O_2:

$\forall x(Triple(x, rdf\!:\!type, O_1\!:\!WhiteWine) \rightarrow$
$Triple(x, rdf\!:\!type, O_2\!:\!Wine) \wedge$
$Triple(x, O_2\!:\!has\_color, "White"))$

3.3 SPARQL Query Rewriting Alghorithm

Similar to other approaches followed in the past, we use graph pattern rewriting in order to translate SPARQL queries, and then we extended this approach for handling data manipulation. In particular, we exploited the alignments between triples structures above described (Section 3.2.2) to rewrite a query in order to be able to run it on different data sets (named as **target** datasets) that employ different ontology asset [21].

3.3.1 Graph Pattern Rewriting Algorithm

The graph pattern rewriting process is used for modifying the RDF graph within a SPARQL query in order to run the same query over different data sets. During the rewriting process, the algorithm matches the head of a rewriting rule and rewrite its body taking into account the eventual variables binding and functional dependencies associated to the rule. A depiction of the RDF graph rewriting between the **`akt`** and the **`kisti`** ontology represented in the Turtle code of the above section, is provided in Figure 2.

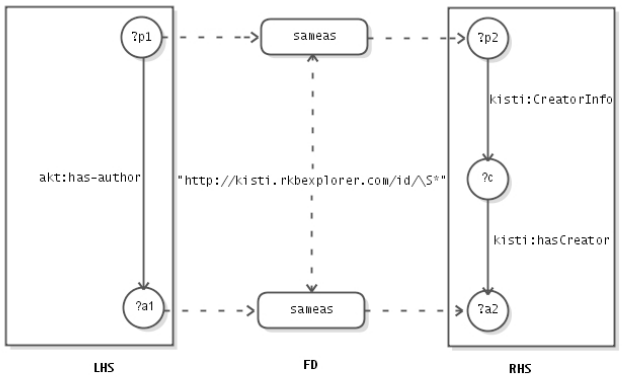


Figure 2: RDF pattern rewriting - example

The alignment depicted in Figure 2 encodes a complex graph pattern rewriting and it is formed of three main parts. On the left of Figure 2 there is the head of the rewriting rule that must be matched against the original triple pattern, the LHS. On the right of Figure 2 there is the pattern to be substituted to the one matched, the RHS. In the middle of Figure 2 lies the functional dependencies between variables that need to be taken into consideration if we have to modify single nodes instead of the whole graph pattern structure. The alignment semantics can then be represented using the three-place notation before reported (note that blank nodes, i.e. `_:p1`, `_:p2`, `_:a1`, `_:a2`, and `_:c`, are interpreted as existentially quantified variables and translated in first-order logic as $?p_1$, $?p_2$ and so on):

$\forall ?p_1 \forall ?a_1 \exists ?p_2 \exists ?c \exists ?a_2 (Triple(?p_1, akt\!:\!has\text{-}author, ?a_1) \rightarrow$
$Triple(?p_2, kisti\!:\!CreatorInfo, ?c) \wedge$
$Triple(?c, kisti\!:\!hasCreator, ?a_2) \wedge$
$?p_2 = sameas(?p_1, "http://kisti.rkbexplorer.com/id/*") \wedge$
$?a_2 = sameas(?a_1, "http://kisti.rkbexplorer.com/id/*"))$

In this formula, the head of the implication (i.e. *LHS*) encodes a graph pattern expressed with the source ontology and the body of the implication (i.e. *RHS*) encodes a graph pattern expressed with the target ontology. The variables in the *LHS* can occur also in the *RHS* of the alignment. Additionally, functional dependencies in the alignment can constraint the values of unbounded variables in an alignment rule, requiring that some variables shall be bounded to the result of the application of a function to already bounded values.

The simple rewriting of the graph pattern however is not enough to support the rewriting of a query to fit a target data set. Different data sets in fact, define a different, and personal, URI space for identifying instances. For example, the URI used in the example (i.e. `http://southampton.rkbexplorer.com/id/person-02686`) represents the person of Nigel Shadbolt within the scope of the data set `http://southampton.rkbexplorer.com`. This does not mean unfortunately that this URI is unique in the whole linked data cloud. In fact, using the **sameas** service[10] we have been able to retrieve over 200 URIs equivalent to the one used in the example[11]. Obviously, if rewriting the structure of the query we would omit to change references to instances URI as well, we will not have the wanted results back when the rewritten query will be run in the target data set.

The `sameas` function used in the example is a wrapper of the online service accessible via REST calls[12]. The **sameas** site provides to the linked data community a service that returns all the URIs that are equivalent to the one given in input. Equivalence between instances are provided by users by annotating URIs with `owl:sameAs` property. The built-in OWL property `owl:sameAs` links individuals together and indicates that two URIs actually refer to the same entity, therefore the individuals have the same "identity".

The function `sameas` returns therefore the first parameter if this is a free variable, or the URI from the equivalence class of the first parameter, using the `owl:sameas` equivalence relation, that matches the regular expression given in input as the second parameter. Returning the first input if it is a free variable is a way of implementing a simple default mechanism that is quite useful to enhance the expressivity of the alignment language.

Summarising, adopting $\approx$ as the symbol for the regular expression string matching and *SAME* as the set of triples contained by the `sameas` service, the *sameas* function is defined as follow:

$$sameas(x, y) = \begin{cases} x & \text{if x is unbounded} \\ z | z \in [x] \wedge z \approx y & \text{otherwise} \end{cases}$$

Where $[x] = \{y \in I | < y, owl\!:\!sameas, x > \in SAME\}$.

Obviously data manipulation functions can come handy in many occasions when integrating heterogeneous data sets. Information can be represented and aggregated in different ways across the semantic web (e.g. different unit measures can be adopted or properties like *address* can be represented all in one value or alternatively each information encoded separately: ZIP code, country, city, street etc.) Representing functional dependencies in a distributed data integration scenario requires more than representing such dependencies. Since the context where the query will be run will be different from the one where the query has been defined and rewritten, we can not assume that every function will be available in every context. The safe assumption here used is that no function has to be known to run a query in a particular context. This means that the functions must be executed when the query is translated, not when they are run, and that requires that all variables declared in a functional dependency section has to be bound to a specific ground value, not to another variable.

In order to rewrite a SPARQL query, according to an ontology alignment, the required steps, in short, are:

1. parse the query and extract its basic graph patterns (or BGP)
2. match every triple of the BGP against the *LHS* of all the entity alignments present in the ontology alignment
3. if the binding obtained can instantiate one of the functional dependencies associated with the entity alignment, apply the function with the binding values and add the resulting value as a binding for the variable
4. substitute the matching triple with the *RHS* of the matched *LHS*, maintaining the bindings obtained by the matching phase and binding all the remaining free variables in *RHS* to a new variable

The binding of all the remaining free variables to a new variable is needed in order to reuse the same alignments in the same rewriting process without introducing unneeded constraints over variables. In detail, the algorithms for translating the basic graph pattern are reported as follows in Algorithm 1 and 2.

In Algorithm 1 the basic graph pattern of a SPARQL query is taken in input; its triples are scanned and considered for a match against the provided alignment. For each triple the algorithm tries to instantiate eventual functions (line 7) and apply the body of the matched rule applying the binding accordingly (line 8). The matching process (line 4), similarly to languages like Prolog, produces a resulting alignment rule (whose *LHS* matches the given triple) plus the binding among variables that satisfy the match.

[10]`http://sameas.org`
[11]The list of URIs has been retrieved in date 16/09/2009
[12]A description of the API available from the **sameas** service can be read at `http://sameas.org/about.php`

Algorithm 1 Algorithm for BGP query rewriting

```
1: function REWRITE($align, bgp$)
2:     $result \leftarrow \oslash$
3:     for all $t \in bgp$ do
4:         $match \leftarrow align.match(t)$
5:         if $match \neq null$ then
6:             for all $triple \in match.getRHS()$ do
7:                 $target \leftarrow instFunction(triple, match, align)$
8:                 $target \leftarrow instPattern(triple, match)$
9:                 $result \leftarrow result \cup target$
10:            end for
11:        else
12:            $result = result \cup t$
13:        end if
14:    end for
15:    return $result$
16: end function
```

The algorithm for instantiating eventual functional dependencies over variables present in the triple is represented in Algorithm 2. Purpose of the *instFunc* function is to exploit declared functional dependencies in order to ensure that ground values in the triple (URIs or literals) are transformed accordingly. In the algorithm, all the variables of a triple coming from a RHS are scanned (line 3 and 4) looking for functional dependencies (line 7). If a functional dependency is present then the parameters are taken into consideration; a parameter within a FD can be either a variable or a ground value. If the parameter is a ground value then it is assigned as input to the function (line 12). If the parameter is a variable that has a binding associated (either to another variable or to a ground value) then the binding is used (line 10). Unbounded variables are left untouched (line 12) because they can be transformed only once the query has returned actual values, thus after the rewriting phase, and are therefore not treated by this approach. Note that this is a pressing issue in a context where the query definition environment and its running environment are strongly decoupled (like in the Semantic Web). Once all the parameters are considered, the function is instantiated and the new value is bounded to the input variable (line 16).

Once the bindings over variables have been enhanced with eventual execution of ground functional dependencies, they are used for instantiate the triples belonging to the body of the alignment rule.

Note that the basic procedure of triples' matching resembles the matching of terms in Prolog language, but with the great simplification here that there are no complex terms in the specification of the alignments, but only variables and instances (URIs and literals). The matching between two nodes in a triple pattern is therefore shortly defined as follows:

$$match(l,r) = \begin{cases} [l/r] & \text{if l} \in \text{Vars} \\ true & \text{if l} \notin \text{Vars} \wedge \text{l = r} \\ false & \text{Otherwise} \end{cases}$$

Where $[l/r]$ is the substitution of the variable l to the value

Algorithm 2 Algorithm for function dependency instantiation

```
1: function INSTFUNCTION($triple, match, a$)
2:     $binding \leftarrow match.getBinding()$
3:     $vars \leftarrow \{v \in triple \wedge v \in Vars\}$    ▷ retrieve all the variables in a triple
4:     for all $var \in vars$ do
5:         $fd \leftarrow match.getFD(var)$    ▷ retrieve the functional dependency for the variable
6:         $param \leftarrow \oslash$
7:         if $fd \neq null$ then
8:             for all $param \in fd.getParameters()$ do
9:                 if $param \in Vars \wedge binding[param] \neq \oslash$ then
10:                    $value \leftarrow binding[param]$
11:                else
12:                    $value \leftarrow param$
13:                end if
14:            end for
15:            $result \leftarrow fd.getFunc().exec(params)$
16:            $binding[var] = result$    ▷ binding is modified
17:        end if
18:    end for
19:    return $triple$
20: end function
```

of the term r (i.e. r itself if r is an unbounded variable, the value of r if r is a variable bounded to a ground value or simply r if it is a ground term). In this definition l is the node present in the LHS of the rule to be matched against r, the node present in the triples of the BGP of the input query. Match over triples just extend this algorithm to subject, predicate and object, returning the union of the substitutions.

3.3.2 Pattern Rewriting Worked Example

A concrete example, following the one of the last section, is provided in this section. Considering the query in Figure 1, the first triple pattern encountered in the BGP is the one asking for:

$Triple(?paper, akt\!:\!has\text{-}author, id\!:\!person\text{-}02686)$

We can match this triple pattern against the LHS part of the following entity alignment present in the knowledge base:

$LHS : Triple(?p_1, akt\!:\!has\text{-}author, ?a_1)$
$RHS : Triple(?p_2, kisti\!:\!CreatorInfo, ?c) \wedge$
$Triple(?c, kisti\!:\!hasCreator, ?a_2)$
$FD :$
$\{?p_2 = sameas(?p_1, "http://kisti.rkbexplorer.com/id/*"),$
$?a_2 = sameas(?a_1, "http://kisti.rkbexplorer.com/id/*")\}$

The matching process will give the following substitutions:

$Triple(?p_1, akt\!:\!has\text{-}author, ?a_1)$
$[?p_1/?paper, ?a_1/id\!:\!person\text{-}02686]$

The only grounded variable of the produced binding is $?a_1$, this allows to fully apply the **sameas** function and retrieve a further binding for a variable belonging to the RHS, a_2: $[?a_2/http://kisti.rkbexplorer.com/id/PER\_0\ldots105046]$.

The variable $?p_1$ is bounded to the variable $?paper$ that is not a ground value, therefore, the result of the **sameas** function is the variable $?paper$ itself, as defined and this produces the new binding $[?p_1/?paper]$.

Applying the substitutions obtained by the previous phases to the body of the entity alignment (i.e. the RHS part) we obtain (the prefix $kisti$ will be used in place of the full URI $"http://kisti.rkbexplorer.com/id/"$ and the prefix id in place of the full URI $"http://southampton.rkbexplorer.com/id/"$):

$Triple(?p_2, kisti{:}CreatorInfo, ?c) \wedge$
$Triple(?c, kisti{:}hasCreator, ?a_2)$
$[?p_1/?paper, ?p_2/?paper, ?c/?new_1, ?a_1/"id{:}person\text{-}02686",$
$?a_2/"kid{:}PER\_0\ldots105047"]$
$\equiv$
$Triple(?paper, kisti{:}CreatorInfo, ?new_1) \wedge$
$Triple(?new_1, kisti{:}hasCreator, "kisti{:}PER\_0\ldots105047")$

Continuing scanning the BGP of the example query, we then meet a similar triple pattern:

$Triple(?paper, akt{:}has\text{-}author, ?a)$

Iterating the same process with the same entity alignment we obtain:

$Triple(?p_2, kisti{:}CreatorInfo, ?c) \wedge$
$Triple(?c, kisti{:}hasCreator, ?a_2)$
$[?p_1/?paper, ?p_2/?paper, ?c/?new_2, ?a_1/?a, ?a_2/?a] \equiv$
$Triple(?paper, kisti{:}CreatorInfo, ?new_2) \wedge$
$Triple(?new_2, kisti{:}hasCreator, ?a)$

The original query in the example is then rewritten as represented in Figure 3.

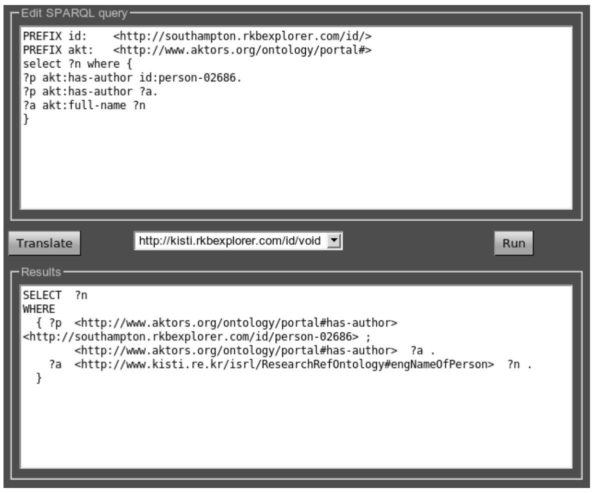
```
PREFIX kid:<http://kisti.rkbexplorer.com/id/>
PREFIX kisti:<http://www.kisti.re.kr/isrl/...#>
SELECT  ?a WHERE  {
    ?p    kisti:hasCreatorInfo  ?_33 .
    ?_33  kisti:hasCreator  kid:PER_0...0105047.
    ?p    kisti:hasCreatorInfo>  ?_38 .
    ?_38  kisti:hasCreator  ?a .
}
```

Figure 3: SPARQL query rewritten

3.4 Implementation

An implementation of the API for managing the alignments described in Section 3.2.1 and 3.2.2 and the algorithm presented in Section 3.3.1 has been implemented using RDF as a syntax for describing the alignments. Jena API[13] has been used for managing the models, and ARQ[14] for accessing the structure of the queries.

The implemented system provides an alignment service from the AKT ontology to the $KISTI$ and DBPedia data set. The system can be accessed either via a web user interface (developed in Google Web Toolkit) and via REST calls. In


Edit SPARQL query

```
PREFIX id:    <http://southampton.rkbexplorer.com/id/>
PREFIX akt:   <http://www.aktors.org/ontology/portal#>
select ?n where {
?p akt:has-author id:person-02686.
?p akt:has-author ?a.
?a akt:full-name ?n
}
```

```
SELECT  ?n
WHERE
  { ?p  <http://www.aktors.org/ontology/portal#has-author>
<http://southampton.rkbexplorer.com/id/person-02686> ;
        <http://www.aktors.org/ontology/portal#has-author>  ?a .
    ?a  <http://www.kisti.re.kr/isrl/ResearchRefOntology#engNameOfPerson>  ?n .
  }
```

Figure 4: Query rewriter user interface

the web user interface the user can write the source SPARQL query (see upper text area in Figure 4) and decide for which data set translate it into. The translated query is then showed in the lower text area (see Figure 4). An additional button can then run the query using the remote SPARQL endpoint provided by the data sets; the results will then be reported in the lower text area.

The system architecture is a simple three-tier architecture that provides services using a Jena back-end for storing and querying the knowledge bases (see Figure 5). The system maintain a simple knowledge base in RDF describing data sets, and their SPARQL endpoints, using the voiD vocabulary[15] (see **voID KB** in Figure 5). Every data set is uniquely identified within the system with an URI. A second knowledge base in RDF is then implemented to describe the ontology alignments (see **Alignment KB** in Figure 5) as presented in Section 3.2.1 and 3.2.2. The execution of the rewritten SPARQL queries is then made via HTTP calls using the endpoints described in their respective voID profiles.

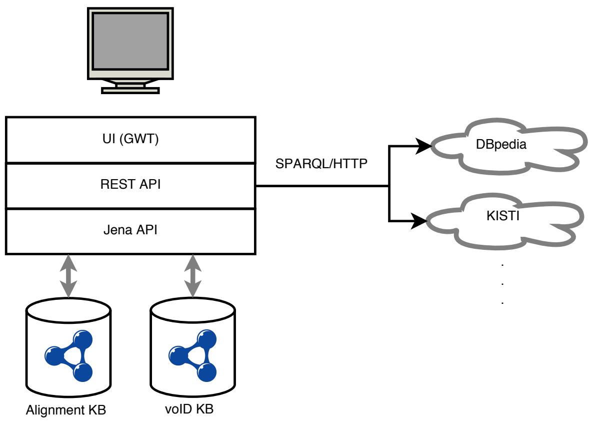


Figure 5: Query rewriter tool architecture

[13]http://jena.sourceforge.net/
[14]http://jena.sourceforge.net/ARQ/
[15]http://rdfs.org/ns/void

The ontology alignment knowledge base implemented for this case counts: 42 alignments (mixed concept and properties alignments) between ECS data set and DBpedia[16]; 24 alignments (mixed concept and properties alignments) between AKT data and KISTI data set[17].

4. CONCLUSIONS AND FUTURE WORK

This paper presented an approach for achieving RDF data integration using query rewriting. The language chosen for the query is SPARQL (W3C recommendation language for querying RDF) and the rewriting phase affects the RDF graph pattern that is the main part of a query definition. The approach followed exploits graph rewriting rules in order to create a new graph pattern that maintain the intended semantics but expressed with the hosting ontology.

The main limitation of the approach is due to an inherent ambiguity of SPARQL language. Query constraints defined within a graph pattern can be expressed also as constraints over variables in the FILTER section of the query. The BGP and the FILTER section are expressed in reference to different models. BGP section describes patterns that candidate solutions shall follow while in the FILTER section the constraints are algebraic expression over values matched in the BGP section. Since the values from the pattern matching phase are treated homogeneously in the evaluation FILTER section, this allows the freedom to express constraints in both sections. As an example, let us consider the query expressed in Figure 1, such query can be expressed also in the form as in Figure 6. The issue is further exacerbated from the fact that usually constraints checking are faster when constraints are expressed within FILTER.

```
PREFIX id:<http://southampton.rkbexplorer.com/id/>
PREFIX akt:<http://www.aktors.org/ontology/portal#>
SELECT DISTINCT ?a WHERE {
  ?paper akt:has-author ?n.
  ?paper akt:has-author ?a.
  FILTER (!(?a = id:person-02686 ) &&
           (?n = id:person-02686))
}
```

Figure 6: SPARQL constraints in FILTER

The problem here is that part of the information needed for a correct rewriting are here put in a part of the query that is not considered by the algorithm because described in a form other than a graph pattern. In order to overcame this difficulty we are currently studying to adapt the approach to the SPARQL algebra [8] that offers the advantage of an homogeneous representation of the whole query (LISP like structures) and a straightforward reference to an underlying semantics representation.

More challenging research scenario would be the one that study how to overcome the structural conflicts between data sets. In fact SPARQL, as a standard language for querying RDF data is still under supported for this purpose, focusing more on how to select and recreate RDF graphs than manipulating non-URI data (i.e. literals as strings, boolean, and numbers). Data processing procedures cannot be integrated in standard SPARQL, cannot be easily exchanged between peers (security in data intensive organizations is crucial) and the trade off between information redundancy and data schema evolution make the integration of web linked data a challenging task to fulfil.

[16]`http://dbpedia.org`

[17]`http://kisti.rkbexplorer.com/`

5. ACKNOWLEDGEMENTS

This research has been supported by the EnAKTing project funded by the Engineering and Physical Sciences Research Council (EPSRC) under the contract EP/G008493/1. The information provided is the sole responsibility of the authors and does not reflect the Council's opinion.

6. REFERENCES

[1] J. Akahani, K. Hiramatsu, and T. Satoh. Approximate query reformulation for multiple ontologies in the semantic web. *NTT Tech. Rev.*, 1(2):83–87, 2003.

[2] S. M. Benslimane, A. Merazi, M. Malki, and D. A. Bensaber. Ontology mapping for querying heterogeneous information sources. *INFOCOMP (Journal of Computer Science)*, Mar. 2008.

[3] N. Bikakis, N. Gioldasis, C. Tsinaraki, and S. Christodoulakis. Querying XML data with SPARQL. In *Proceedings of the* 20th *International Conference on Database and Expert Systems Applications*, pages 372–381, 2009.

[4] P. Bouquet, F. Giunchiglia, F. van Harmelen, L. Serafini, and H. Stuckenschmidt. C-OWL: Contextualizing ontologies. *The SemanticWeb - ISWC 2003*, pages 164–179, 2003.

[5] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. What is query rewriting? In *Proceedings of the* 7th *International Workshop on Knowledge Representation meets Databases*, pages 17–27, 2000.

[6] A. Chebotko, S. Lu, H. M. Jamil, and F. Fotouhi. Semantics preserving SPARQL-to-SQL query translation for optional graph patterns. Technical report, Wayne State University, Department of Computer Science, November 2007.

[7] H. Chen. Rewriting queries using view for RDF/RDFS-based relational data integration. *Distributed Computing and Internet Technology*, 3816:243–254, December 2005.

[8] R. Cyganiak. A relational algebra for SPARQL. Technical Report `http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html`, HP Laboratories Bristol, September 2005.

[9] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs). RFC 3987, Jan. 2005.

[10] M. Ehrig and Y. Sure. *Professional Knowledge Management*, volume 3782 of *LNCS*, chapter Ontology Mapping by Axioms (OMA), pages 560–569. Springer Berlin / Heidelberg, 2005.

[11] J. Euzenat. An API for ontology alignment. In *Proceedings of the* 3rd *International Semantic Web Conference*, pages 698–712, 2004.

[12] J. Euzenat. Algebras of ontology alignment relations. In *Proceedings of the* 7th *International Semantic Web Conference*, pages 387–402, 2008.

[13] J. Euzenat, A. Polleres, and F. Scharffe. Processing ontology alignments with SPARQL. In *Proceedings of*

the International Conference on Complex, Intelligent and Software Intensive Systems, pages 913–917, 2008.

[14] J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer, Berlin, Heidelberg, 2007.

[15] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. Integrating and accessing heterogeneous information sources in TSIMMIS. In *Proceedings of the AAAI Symposium on Information Gathering*, pages 61–64, 1995.

[16] H. Glaser, A. Jaffri, and I. Millard. Managing co-reference on the semantic web. In *WWW2009 Workshop: Linked Data on the Web (LDOW2009)*, April 2009.

[17] F. Goasdoué and M.-C. Rousset. Answering queries using views: A KRDB perspective for the semantic web. *ACM Trans. Internet Techn.*, 4(3):255–288, 2004.

[18] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *Proceedings of the* 19^{th} *International Conference on Data Engineering*, pages 505–516, 2003.

[19] H. Halpin. Identity, Reference, and Meaning on the Web. In *Proceedings of the Workshop on Identity, Meaning and the Web (IMW06)*, 2006.

[20] P. Hayes. RDF semantics. Technical Report `http://www.w3.org/TR/rdf-mt/`, W3C, February 2004.

[21] Y. Jing, D. Jeong, and D.-K. Baik. SPARQL graph pattern rewriting for OWL-DL inference queries. *Knowledge and Information Systems*, 20:243–262, 2009.

[22] J. Li, J. Tang, Y. Li, and Q. Luo. RiMOM: A dynamic multistrategy ontology alignment framework. *IEEE Trans. Knowl. Data Eng.*, 21(8):1218–1232, 2009.

[23] K. Makris, N. Bikakis, N. Gioldasis, C. Tsinaraki, and S. Christodoulakis. Towards a mediator based on OWL and SPARQL. In *Proceedings of the* 2^{nd} *World Summit on the Knowledge Society*, pages 326–335, 2009.

[24] A. D. Preece, K. ying Hui, W. A. Gray, P. Marti, T. J. M. Bench-Capon, D. M. Jones, and Z. Cui. The KRAFT architecture for knowledge fusion and transformation. *Knowl. Based Syst.*, 13(2-3):113–120, 2000.

[25] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. Technical Report `http://www.w3.org/TR/rdf-sparql-query/`, W3C, January 2008.

[26] B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *Proceedings of the* 5^{th} *European Semantic Web Conference*, pages 524–538, 2008.

[27] S. Simon. A SPARQL semantics based on datalog. In *Proceedings of the* 30^{th} *annual German conference on Advances in Artificial Intelligence*, pages 160–174, 2007.

[28] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceeding of the* 17^{th} *International Conference on World Wide Web*, pages 595–604, 2008.

[29] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, 1992.