A Data Mashup Language for the Data Web

Mustafa Jarrar
University of Cyprus
mjarrar@cs.ucy.ac.cy

Marios D. Dikaiakos
University of Cyprus
mdd@cs.ucy.ac.cy

ABSTRACT

This paper is motivated by the massively increasing structured data on the Web (Data Web), and the need for novel methods to exploit these data to their full potential. Building on the remarkable success of Web 2.0 mashups, this paper regards the internet as a database, where each web data source is seen as a table, and a mashup is seen as a query over these sources. We propose a data mashup language, which allows people to intuitively query and mash up structured and linked data on the web. Unlike existing query methods, the novelty of MashQL is that it allows people to navigate, query, and mash up a data source(s) without any prior knowledge about its schema, vocabulary, or technical details. We even do not assume even that a data source should an online or inline schema. Furthermore, MashQL supports query pipes as a built-in concept, rather than only a visualization of links between modules.

Published as:
Mustafa Jarrar and Marios D. Dikaiakos: *A Data Mashup Language for the Data Web*. Proceedings of LDOW, at WWW'09, ACM, 2009.

1. INTRODUCTION AND MOTIVATION

In this short article we propose a data mashup approach in a graphical and Yahoo Pipes' style. This research is still a work in progress, thus please refer to [13] for the latest findings.

In parallel to the continuous development of the hypertext web, we are witnessing a rapid emergence of the Data Web. Not only the amount of social metadata is increasing, but also many companies (e.g., Google Base, Upcoming, Flicker, eBay, Amazon, and others) started to make their content freely accessible through APIs. Many others (see linkeddata.org) are also making their content directly accessible in RDF and in a linked manner [3]. We are also witnessing the launch of RDFa, which allows people to access and consume HTML pages as structured data sources.

This trend of structured and linked data is shifting the focus of web technologies towards new paradigms of *structured-data retrieval*. Traditional search engines cannot serve such data because their core design is based on keyword-search over unstructured data. For example, imagine how would be the results when using Google to search a database of job vacancies, say "well-paid research-oriented job in Europe". The results will not be precise or clean, because the query itself is still ambiguous although the underlying data is structured. People are demanding to not only retrieve job links but also want to know the starting date, salary, location, and may render the results on a map.

Web 2.0 mashups are a first step in this direction. A mashup is a web application that consumes data originated from third parties and retrieved via APIs. For example, one can build a mashup that retrieves only well-paid vacancies from Google Base and mix it with similar vacancies from LinkedIn. The problem is that building mashups is an art that is limited to skilled programmers. Although some mashup editors have been proposed by the Web 2.0 community to simplify this art (such as Google Mashups, Microsoft's Popfly, IBM's sMash, and Yahoo Pipes), however, what can be achieved by these editors is limited. They only focus on providing encapsulated access to *some* APIs, and still require programming skills. In other words, these mashup methods are motivating for -rather than solving- the problem of structured-data retrieval. To expose the massive amount of structured data to its full potential, people should be able to query and mash up this data easily and effectively.

Position: To build on the success of Web 2.0 mashups and overcome their limitation, we propose to regard the web as a database, where each data source is seen as a table, and a mashup is seen as a query over one or multiple sources. In other words, instead of developing a mashup as an application that access structured data through APIs, this art can be simplified by regarding *a mashup as a query*. For example, instead of developing a "program" to retrieve and fuse certain jobs from Google Base and Jobs.ac.uk, this program should be seen as a data query over two remote sources. Query formulation (i.e., mashup development or data fusion) should be fast and should not require any programming skills.

Challenges: Before a user formulates a query on a data source, she needs to know how the data is structured, and what are the labels of the data elements, i.e., the schema. Web users are not expected to investigate "what is the schema" each time they search or filter structured information. This issue is particularly more difficult in case of RDF and linked data. RDF data may come without a schema\ontology, and if exists, the schema is mixed up with the data. In addition, as RDF data is a graph, one have to manually navigate this graph in order to formulate a query about it. Imagine large and multiple linked data sources, with diverse content and vocabularies, how you would manage to understand the data structure, inter-relationships, namespaces, and the unwieldy labels of the data elements. *In short, formulating queries in open environments, where data structures and* *vocabularies are unknown in advance, is a hard challenge, and may hamper building data mashups by non-IT people.*

To allow people to query and mash up data sources intuitively, we propose a data mashup language, called MashQL. The main novelty of MashQL is that it allows non IT-skilled people to query and explore one (or multiple) RDF sources without any prior knowledge about the schema, structure, vocabulary, or any technical details of these sources. To be more robust and cover most cases in practice, we even do not assume that a data source should have -an offline or online- schema\ontology at all. In the background, MashQL queries are translated into and executed as SPARQL queries.

Paper organization: Before presenting MashQL, in the next section we overview the art of query formulation, which has been studied by different research communities. We present MashQL in section 3, and in section 4 we introduce the notion of query pipes. The implementation of MashQL and a three use cases are presented in section 5 and 6 respectively. The coverage and the limitations of MashQL and its future directions are discussed in section 7.

2. RELATED WORK

Several approaches have been proposed by the DB community to query structured data sources, such as *query-by-example* [23] and *conceptual queries* [4,6,17]. However, none of these approaches was used by casual users. This is because they still assume knowledge about the relational/conceptual schema. Among these, we found ConQuer [4] has some nice features, specially the tree structure of queries, but it also assumes one to start from the schema. In the natural language processing community, it has been proposed to allow people to write *queries as natural language sentences*, and then translate these sentences into a formal language (SQL [15] or XQuery [16]). However, these approaches are challenged with the language ambiguity and the “free mapping” between sentences and data schemes.

This topic started to receive a high importance within the Semantic Web community. Several approaches (GRQL [1], iSPARQL [11], NITELIGHT [19] and RDFAuthor [18]) are proposing to *represent triple patterns graphically* as ellipses connected with arrows. However, these approaches assume advanced knowledge of RDF and SPARQL. Other approaches use *Visual Scripting Languages* (e.g., SPARQLMotion [21] and Deri Pipes [22]), by visualizing links between query modules; *but a query module merely is a window containing a SPARQL script in a textual form*. These approaches are inspired by some industrial mashup editors such as Popfly, sMash, and Yahoo Pipes. These industry editors provide a nice visualization of APIs’ interfaces and some operators between them. However, when a user needs to express a query over structured data, she needs to use the formal language of that editor, such as YQL for Yahoo Pipes. Although MashQL visualizes links between query modules, similar to Yahoo Pipes and other Mashup editors, but the main purpose of MashQL is *to help people to formulate what is inside these query modules*.

Differently from the above Web 2.0 mashup editors, a more sophisticated editor has been proposed in [8], called MashMaker. It is a functional programming environment that allows one to mashup web content in a spreadsheet-style user interface. Like a spreadsheet, MashMaker stores every value that is computed in a single, central data structure. MashMaker is not comparable with MashQL since it cannot serve as a query language by it is own.

In XML databases, the Lore query language [9] has been proposed to allow people to query XML data graphically, and without prior knowledge about the data. Lore assumes that data is represented as a graph, called EOM, which is close to RDF. The difference between Lore and MashQL is not only the intuitiveness and expressivity, but essentially, MashQL does not assume the data graph to have a certain schema, however, Lore assumes that a data graph should have a dataguide, which is a computed summary of the data, i.e. play the role of a schema.

More about query formulations scenarios and (which scenario is more intuitive to the casual user) can be found in a recent usability study in [14]. It concluded that a query language should be close to natural language and graphically intuitive, and it should not assume knowledge about the data source.

3. THE MASHQL LANGUAGE

The main goal of MashQL is to allow people to mash up and fuse data sources easily. In the background MashQL queries are automatically translated into and executed as SPARQL queries. Without prior knowledge about a data source, one can navigate this source and fuse it with another source easily. To allow people to build on each other’s results MashQL supports query pipes as a built-in concept. The example below shows two web data sources and a SPARQL query to retrieve “the book titles authored by Lara and published after 2007”. The same query in MashQL is shown in Figure 2. The first module specifies the query input, and the second module specifies the query body. The output can be piped into a third module (not shown here), which renders the results into a certain format (such as HTML,XML or CSV), or as RDF input to other queries. Notice that in this way, one can easily build a query to fuse the content of two sources in a linked manner [3].

http://Site1.com/RDF

```
:a1 :Title “Web 2.0”
:a1 :Author “Hacker B.”
:a1 :Year  2007
:a1 :Publisher “Springer”
:a2 :Title “Web 3.0”
:a2 :Author “Smith B.”
```

http://Site2.com/RDF

```
:4 :Title “Semantic Web”
:4 :Author “Tom Lara”
:4 :PubYear  2005
:5 :Title “Web services”
:5 :Author “Bob Hacker”
```

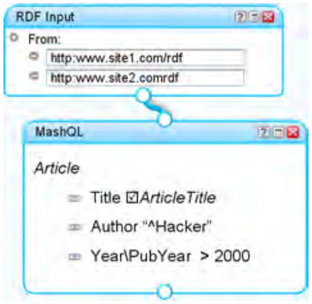
Query:

```
PREFIX S1: <http://site1.com/rdf>
PREFIX S2: <http://site1.com/rdf>
SELECT ? ArticleTitle
FROM <http://site1.com/rdf>
FROM <http://site2.com/rdf>
WHERE {
 {{?X S1:Title ?ArticleTitle}UNION
 {?X S2:Title ?ArticleTitle}}
 {?X S1:Author ?X1} UNION {?X S2:Author ?X1}
 {?X S1:PubYear ?X2} UNION {?X S2:Year ?X2}
 FILTER regex(?X1, “^Hacker”)
 FILTER (?X2 > 2000)}
```

Results:

ArticleTitle
Web 2.0

Figure 1. An example of a SPARQL query.



RDF Input

From:

http:www.site1.com/rdf

http:www.site2.comrdf

MashQL

Article

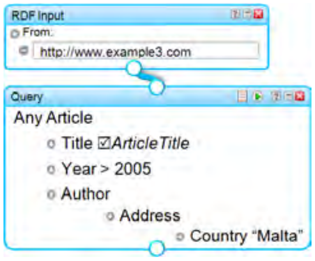
Title ☑*ArticleTitle*

Author "^Hacker"

Year\PubYear > 2000

Figure 2. An example of MashQL query.

The intuition of MashQL is described as the following: Each query *Q* is seen as a tree. The root of this tree is called the *query subject* (e.g. Article), denoted as *Q(S)*, which is the subject matter being inquired. Each branch of the tree is called a *restriction R* and is used to restrict a certain property of the query subject, $Q(S) \coloneqq R_1 \text{ AND } \ldots \text{ AND } R_n$. Branches can be expanded to allow sub trees (called *query paths*), which enable one to navigate the underlying dataset. In this case, the object in the restriction is considered the subject of its sub query. As Figure 3 shows, the query retrieves the title of every article, published after 2005, and written by an author, who has an address, this address has a country called Cyprus.



RDF Input

From:

http://www.example3.com

Query

Any Article

Title ☑*ArticleTitle*

Year > 2005

Author

Address

Country "Malta"

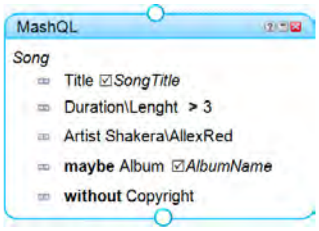
```
PREFIX S1: <http://www.example.com>
SELECT ?ArticleTitle
FORM < http://www.example.com?
WHERE { ?X1 rdf:type :Article.
        ?X1 S1:Title ?ArticleTitle.
        ?X1 S1:Year ?X2.
        FILTER (?X2 > 2005).
        ?X1 S1:Author ?X3.
        ?X3 S1:Address ?X4.
        ?X4 S1:Country ?X5.
        FILTER regex(?X5, “Malta”)}
```

Figure 3. A query involving paths, and its mapping into SPARQL.

Formulating queries in MashQL is designed to be an interactive process, by which the complexity of understanding data structures is moved to the query editor. Users only use drop-down lists to express their queries.

The query subject is selected from a list generated dynamically from, either: (1) the set of the subject-types in the dataset; (2) or the union of all subject and object identifiers in the dataset; users can also choose to (3) introduce their own label; in this case the label is seen as a variable and displayed in *italic*. The default subject is the variable "Anything". To add a restriction, the list of properties (e.g., Title, Author) is generated, depending on the chosen subject. Users may then select a filter (e.g., Equals, Contains, Between, etc.), or select an object identifier from a list, which is then generated from the set of the possible objects identifies, depending on the previous selections. Furthermore, users select to expand the tree to declare a query path. The projection symbol ☑ can be used before a variable to indicate that it will be returned in the results[1]. In short, while interacting with the editor, the editor queries the dataset in the background in order to generate the next list depending on the previous selections. In this way, people can navigate a graph without prior knowledge about it.

Similar to SPARQL, all restrictions in MashQL are considered necessary when evaluating a query. However, if a restriction is prefixed with "maybe", it is considered optional; and, if it is prefixed with "without" it is considered unbound (see Figure 3). MashQL supports also union (denoted as "\") between objects, predicates, subjects, and queries; as well as, a type operator ("Any"), Inverse predicates, datatype and language tags, and many objects filters.



MashQL

Song

Title ☑*SongTitle*

Duration\Lenght > 3

Artist Shakera\AllexRed

maybe Album ☑*AlbumName*

without Copyright

```
PREFIX a: <http:www.example.nam.com>
PREFIX S1: <http:www.example.si.com>
SELECT ?SongTitle, ?AlbumName
FROM <http:www.example.si.com>
WHERE {?Song S1:Title ?SongTitle.
  {{?Song S1:Duration ?X1}
UNION {?Song a:Length ?X1}}
  FILTER (?X1 > 3).
  {{?Song S1:Artist S1:Shakira}
UNION {?Song S1:Artist S1:AxelleRed}}
 OPTIONAL{?Song S1:Album ?AlbumName}.
  OPTIONAL{?Song S1:Copyright ?X2}.
  FILTER (!Bound(?X2)).}
```

Figure 4. A query involving optional and negative restrictions.

4. THE NOTION OF QUERY PIPES

To deploy MashQL in an open world some challenges might be faced. This section overviews these challenges (from a query formulation viewpoint) and introduces the notion of *query pipes*.

As discussed earlier, one may create a mashup and redirect its output to another mashup. We call the chain of queries that connect to each other in this way as *pipe*. Allowing people to formulate query pipes is not merely a visualization of links between query modules, but when compiling a pipe (i.e., translating it into SPARQL), some issues should be considered.

First: Translating MashQL into SPARQL SELECT statements is not enough, because the SELECT statement produces the results in a tabular form. To allow queries to input each other (especially

[1] Some issues are lengthy to illustrate here. For example, when a user moves the mouse over a restriction, it gets the *editing mode* and all other restrictions get the *verbalize mode* (i.e., all boxes and lists are made invisible, but the verbalization of their content is generated and displayed instead). This is not only to make the readability of the queries closer to natural language, but also to allow users to validate whether what they did is what they intended. The editor also detects and normalizes namespaces: find similar URLs and hide them when necessary. For example, when two properties originating from different data sources have the same URL, their namespaces are found and hided.

for producing linked data), the results of a query should be formed as a graph. In SPARQL, the CONSTRUCT statement produces a graph, but then one needs to manually specify how this graph should be produced. To overcome this, we propose the construct (CONSTRUCT *). This is not part of the standard SPARQL but has been proposed also by others to be included in the next version of the standard [20]. In MashQL, the CONSTRUCT * means *retrieves all triples involved in the query conditions and satisfy them*. For example, suppose the query in Figure 2 is piped into another, its CONSTRUCT * translation will retrieve `{<:b1 :Title “Linked Data”>,<:b1 :Author “Lara T.”>,<:b1 :Year 2007>}`. When compiling a pipe of queries, If the output of a query is directed as input to another query, a CONSTRUCT * statement will be generated, otherwise, a SELECT statement will be generated.

Second: When executing a SPARQL query, all query engines assume that the queried data is stored locally; otherwise, this data must be downloaded and stored at the engine-side before the execution process starts. The time complexity of executing a query on local data is usually fast[2]; however, the bottleneck will be the downloading time. In case the input of a query is an output to another query (i.e., in case of query pipes) the problem will be even more difficult, as queries will be calling each other. Furthermore, it is also possible that users (intentionally or by mistake) end up with query loops (e.g. $Q_1 \rightarrow Q_2 \rightarrow Q_3 \rightarrow Q_1$), which may cause computational overheads. To face this challenge, MashQL allows users to materialize the results of their queries/pipes and decide their refreshing strategies, as follows:

The results of a query (called *derived source*) are stored physically and deployed as a concrete RDF source. Primal input sources (called *base sources*) are also cached for performance purposes. Given a query Q over a set of base or derived sources $\{D_1,..,D_m\}$, the results of this query is denoted as $D = Q(D_1,..,D_m)$, and $D \notin \{D_1,..,D_m\}$. We define a *Pipe* as an *acyclic* chain of queries, where the result of a query is an input to the next. The chain of the queries that derives D is denoted as the pipe $P(D)$.

We call the problem of keeping a pipe up-to-date, the *pipes consistency*. Let D be the results of a query $Q(D_1,..,D_m)$, and T the latest time the set $\{D_1,..,D_m\}$ has been changed. Then, D is consistent at T if $D=Q(D_1,..,D_m)$. To maintain pipes consistency, two updating strategies are used: Query auto-refresh and Pipe auto-refresh. MashQL maintains for each base or derived source D a timestamp of its last update R_D^T and an auto-refresh time interval R_D^A; and for each query Q a timestamp of its previous successful execution R_Q^T and an auto-refresh interval R_Q^A.

Query auto-refresh: Each query will be automatically executed if its auto-refresh interval expires and one of its inputs is updated. Let Q_i be a query over a set of sources $\{D_1,..,D_m\}$, and T is a given time. Q_i will be re-executed *if* $(R_{Qi}^T + R_{Qi}^A) \leq T$ and $(R_{Qi}^T < R_{Dj}^T)$, where $1 \leq j \leq m$.

Pipe auto-refresh: Each pipe $P(D)$ is automatically refreshed if R_D^A expires. This implies re-executing the chain of queries in this pipe. Let $P(D)$ be a pipe, $D=Q_n(D_1,..,D_m)$, and T is a given time. *If* $(R_D^T+R_D^A) \leq T$, then each i^{th} query in $P(D)$ is executed if $(R_{Qi}^T < R_{Dj}^T)$, where $1 \leq j \leq m$ for Q_i, and $1 \leq i \leq n$. Queries in $P(D)$ are executed from the bottom to the topmost, or recursively as $P(P(D_1),\ldots,P(D_m))$.

As argued in the data warehousing literature [2,24] an efficient refreshing strategies is the *incremental updates*, which suggests that if a base source receives new transactions, only these transactions are transformed and the affected queries are refreshed. This strategy is still an open research issue for RDF in an open world [7], because RDF data and queries are developed and maintained autonomously by different people.

5. IMPLEMENTATION

First: we have developed an online mashup editor, which will be publically available next month. Similar to creating feed mashups in Yahoo Pipes, MashQL users can query and fuse data sources and the output of their queries can be redirected as input to other queries. In the background, Oracle 11g is used for storing and querying RDF. When a user specifies a data source(s) as input, it is bulk-loaded to the Oracle’s semantic technology tables. MashQL queries are also translated into Oracle’s SPARQL. While interacting with the editor to formulate a query, the editor performs some background queries through AJAX. Each published query is given a URL. Calling this URL means executing this query and getting its results back.

Second: We started to also develop a Firefox add-on in order to allow people develop mashups at the client side. The opened pages -in the browser tabs- are automatically selected as input sources, and at the left-side panel a mashup can be created. The results are rendered by the browser in a new tab. The idea is to allow web pages that embed RDF triples (i.e., RDFa or microformats) to be queried and mashed up. For example, one will be able to compose his publication list from Google Scholar, DBLP, ACM, and CiteSeer; or, filter all video lectures given by Berners-Lee from YouTube and VedioLectures. Because the mentioned web sites do not support RDFa yet, one can mine/distil the RDF triples, using third party services such as triplr.org, buzzword.org.uk, wandora.org or Dapper.

[2] A query with medium size complexity over a large dataset takes one or few seconds [5].

6. USE CASES

This is section we present two hypothetical use cases to illustrate using MashQL for developing data mashups.

6.1 Use case: Retailer

Fnac is a large retailer of cultural and consumer electronics products. When a new product arrives to Fnac, it has to be entered to the inventory database. This is usually done by scanning the barcode on each product, and then manually filling the product specifications. Furthermore, as Fnac trades in many countries, their product specifications have to be translated into several languages. To save time entering and translating information manually, Fnac decided to reuse the product data specifications (and their translation) that are produced at the factory side. For example, suppose Fnac received three packages from Cannon, Alfred, and IMDB. Fnac would like to scan the barcode of the received products and then get their specifications directly from the online catalogues of those suppliers. In Figure 5 we show samples of online product catalogues of the three suppliers (we assume they are published in RDFa). Figure 6 illustrates a query that Fnac built to look up the multilingual titles of three products. This query is a mashup of three RDF data sources with a user-input of three barcode numbers. The query takes each of these barcodes and finds the English and French titles. Notice that Fnac assumed that short titles provided by Cannon are in English, thus, they are joined with the other titles that are tagged with "@en". See the retrieved results in Figure 8. In this same way, a barcode reader could be connected with user-input module, to retrieve the specifications (which could be stored at the supplier side) each time a product is scanned.

http:www.cannon/products/rdf

```
_:P1 :ShortName “CanScan 4400F”
_:P1 :FullName “Canon CanoScan
       4400F Color Image Scanner”
_:P1 :Producer “Canon”
_:P1 :ShippingWeight> “4 pounds”
_:P1 :Barcode 9780133557022
_:P2 :ShortName “PowerShot SD100”
_:P2 :FullName “Canon PowerShot
   SD10007.1MP Camera 3x Zoom”
_:P2 :Producer “Canon”
_:P2 :ShippingWeight> “2 pounds”
_:P2 :Barcode 9781143557532
```

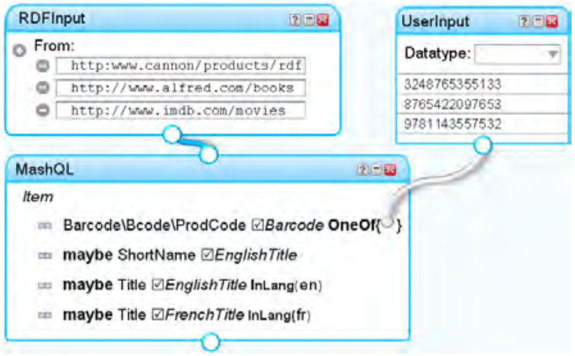
http://www.alfred.com/books

```
<:B1> :Type <:Book>
<:B1> :Title “The Prophet”@en
<:B1> :Title “Le prophète”@fr
<:B1> :BCode 8765422097653
<:B1> :Authors “Kahlil Gibran”
<:B1> :ISBN-10 0394404289
<:B3> :Type <:Book>
<:B3> :Title “Alfred Nobel”@en
<:B3> :Title “Alfred Nobel”@fr
<:B3> :BCode 75639898123
<:B3> :Authors “Kenne Fant”
<:B3> :ISBN- 0531123286
```

http://www.imdb.com/movies

```
_:1 rdf:Type <:Movie>
_:1 :Title “All about my mother”@en
_:1 :Title “Tout sur ma mère”@fr
_:1 :ProdCode 3248765355133
_:1 :NumberOfDiscs: 1
_:2 rdf:Type <:Movie>
_:2 :Title “Lords of the rings”@en
_:2 :Title “Seigneur des anneaux”@fr
_:2 : ProdCode 4852834058083
_:2 :NumberOfDiscs: 3
```

Figure 5. Sample of RDF data about products.



RDFInput

From:

http:www.cannon/products/rdf
http://www.alfred.com/books
http://www.imdb.com/movies

UserInput

Datatype:

3248765355133
8765422097653
9781143557532

MashQL

Item

Barcode\Bcode\ProdCode ☑*Barcode* **OneOf{** }
maybe ShortName ☑*EnglishTitle*
maybe Title ☑*EnglishTitle* **InLang**(en)
maybe Title ☑*FrenchTitle* **InLang**(fr)

Figure 6. A mashup of product titles from different resources.

```
PREFIX s1: <http:www.cannon/products/rdf>
PREFIX s2: <http://www.alfred.com/books>
PREFIX s3: <http://www.imdb.com/movies>
SELECT ?Barcode ?EnglishTitle ?FrenchTitle
FROM <http:www.cannon/products/rdf>
FROM <http://www.alfred.com/books>
FROM <http://www.imdb.com/movies>
WHERE{
 {{?x s1:Barcode ?Barcode}  UNION {?x s2:Bcode ?Barcode}
    UNION  {?x s3:Prodcode ?Barcode}}
 FILTER (regex(?Barcode, “9781143557532”)  ||
         regex(?Barcode, “8765422097653”)  ||
         regex(?Barcode, “3248765355133)”).
{OPTIONAL {?x s1:ShortName ?EnglishTitle}}  UNION
{{OPTIONAL {?x s1:Title ?EnglishTitle}} UNION
 {OPTIONAL {?x s2:Title ?EnglishTitle}}
 FILTER (lang(?EnglishName) = ”en”)}
 {{OPTIONAL {?x s1:Title ?FrenchTitle}} UNION
  {OPTIONAL {?x s2:Title ?FrenchTitle}}
 FILTER (lang(?FrenchTitle) = ”fr”)}}
```

Figure 7. The SPARQL equivalent of Figure 6.

Barcode	EnglishTitle	FrenchTitle
9781143557532	CanScan 4400F	
8765422097653	The Prophet	Le prophète
3248765355133	All about my mother	Tout sur ma mère

Figure 8. Retrieved product titles.

6.2 Use case: Citations List

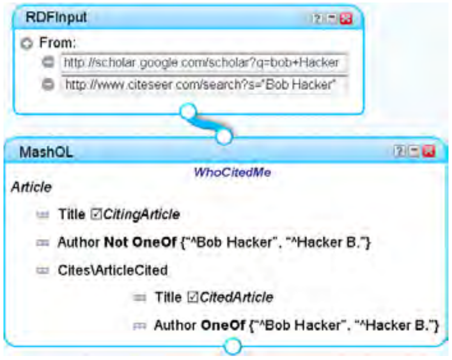
Bob would like to compile the list of articles that cited his articles (excluding what he cited himself). He built a mashup using MashQL to mix his citations retrieved from both Google Scholar and CiteSeer, and then filter out the self-citations. First, he performed a keyword search (“Bob Hacker”) on both Google Scholar and CiteSeer3. Figure 9 shows a sample of the extracted RDF triples. Bob’s MashQL query is shown in Figure 10, and its SPARQL equivalent in Figure 11. In this query, Bob wrote: retrieve every article that has a title (call it CitingArticle), has an

[3] Similar to the previous use case, we assume that both Google Scholar’s and CiteSeer’s render their search results in RDFa (i.e. the RDF triples are embedded in HTML), as many companies started to do nowadays. However, Bob can also use a third party’s service (e.g. triplify.org) to extract triples from HTML pages.

author that does not contain "Bob Hacker" or "Hacker B.", and cites another article that has a title (call it CitedArticle), and has an author that contains "Bob Hacker" or "Hacker B.". Figure 12 shows the result of this query.

http://scholar.google.com/scholar?q=bob+Hacker	**http://www.citeseer.com/search?s="Bob Hacker"**
`<g:3> :Title  “Prostate Cancer”`	`_:1 :Title “Prostate Cancer”`
`<g:3> :Author “Hacker B.,Hacker A.”`	`_:1 :Author “Hacker B., Hacker A.”`
`<g:4> :Title “Best and Worst Lifestyles”`	`_:2 :Title  “Protocols in Molecular Biology”`
`<g:4> :Atuhor “Bob Hacker”`	`_:2 :Atuhor  “Bob Hacker”`
`<g:4> :Cites <g:3>`	`_:2 :ArticleCited  _:1`
`<g:7> :Title “Protein Categories”`	`_:3 :Title  “Cancer Vaccines”`
`<g:7> :Atuhor “Bob Smith”`	`_:3 :Atuhor  “Eve Lee, Bob Hacker”`
`<g:7> :Cites <g:3>`	`_:4 :Title “Overview about Systems Biology”`
`<g:7> :Cites <g:4>`	`_:4 :Atuhor  “Tom Lara”`
`<g:8> :Title “Cancer Vaccines”`	`_:4 :ArticleCited  _:1`
`<g:8> :Atuhor “Alice Hacker”`	`_:4 :ArticleCited  _:2`
`<g:8> :Cites <g:3>`	

Figure 9. Sample of RDF data about Bob’s articles.



RDFInput

From:

- http://scholar.google.com/scholar?q=bob+Hacker
- http://www.citeseer.com/search?s="Bob Hacker"

MashQL

WhoCitedMe

Article

Title ☑*CitingArticle*

Author **Not OneOf** {"^Bob Hacker", "^Hacker B."}

Cites\ArticleCited

Title ☑*CitedArticle*

Author **OneOf** {"^Bob Hacker", "^Hacker B."}

Figure 10. A mashup of citation from different sites.

```
PREFIX s1: http://scholar.google.com/scholar?q=bob+Hacker
PREFIX s2: http://www.citeseer.com/search?s=“Bob Hacker
SELECT CitingArticle? ?CitedArticle
From <http://scholar.google.com/scholar?q=bob+Hacker>
From <http://www.citeseer.com/search?s=“Bob Hacker”>
WHERE {
  {{?X1 s1:Title ?CitingArticle} UNION
   {?X1 s2:Title ?CitingArticle}}
  {{?X1 s1:Author ?X2} UNION {?X1 s2:Author ?X2}}
  {{?X1 s1:Cites ?X3} UNION {?X1 s2:ArticleCited ?X3}}
  {{?X3 S1:Title ?CitedArticle} UNION
  {?X3 S2:Title ?CitedArticle}
  {{?X3 s1:Author ?X4} UNION {?X3 s2:Author ?X4}}
  FILTER    (regex(?X2,”^Bob    Hacker”)||regex(?X2,”^Hacker
B.”))}
  FILTER Not(regex(?X4,”^Bob Hacker”) ||
             regex(?X4,”^Hacker B.”)) }
```

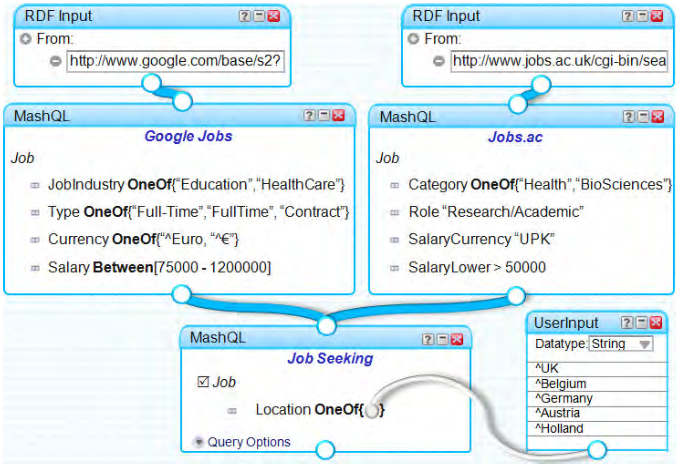
Figure 11. The SPARQL equivalent of Figure 10.

CitingArticle	CitedArticle
Protein Categories	Prostate Cancer
Protein Categories	Best and Worst Lifestyles
Cancer Vaccines	Prostate Cancer
Overview about Systems Biology	Prostate Cancer
Overview about Systems Biology	Protocols in Molecular Biology

Figure 12. The query results.

6.3 Use case: Job Seeking

Bob has a PhD in bioinformatics. He is looking for a full-time, well paid, and research-oriented job in some European countries. He spent an enormous amount of time searching different job portals, each time trying many keywords and filters. Instead, Bob used MashQL to find the job that meets his specific preferences. Figure 13 shows Bob’s queries on Google Base and on Jobs.ac.uk. First, he visited Google Base and performed a keyword search (bioinformatics OR "computational biology" OR "systems biology" OR e-health); he copied the link of the retrieved results from Google (which are in rendered in RDFa) into the RDFInput module; and then created a MashQL query on these results. He performed a similar task to query Jobs.ac.uk. The third MashQL module in Figure 13, mixes the results of the above two queries and filters them based on location preferences (provided in the UserInput module). The SPRQAL equivalent to Bob’s MashQL query is shown in Figure 14.



RDF Input

From:

- http://www.google.com/base/s2?

RDF Input

From:

- http://www.jobs.ac.uk/cgi-bin/sea

MashQL

Google Jobs

Job

JobIndustry **OneOf**{“Education”,“HealthCare”}

Type **OneOf**{“Full-Time”,“FullTime”, “Contract”}

Currency **OneOf**{“^Euro, “^€”}

Salary **Between**[75000 - 1200000]

MashQL

Jobs.ac

Job

Category **OneOf**{“Health”,“BioSciences”}

Role “Research/Academic”

SalaryCurrency “UPK”

SalaryLower > 50000

MashQL

Job Seeking

☑ *Job*

Location **OneOf**{ }

Query Options

UserInput

Datatype: String

^UK
^Belgium
^Germany
^Austria
^Holland

Figure 13. Bob’s mashup of jobs.

```
CONSTRUCT *
WHERE {?Job :JobIndustry ?X1;
            :Type ?X2;
            :Currency ?X3;
            :Salary ?X4.
FILTER(?X1=“Education”||
       ?X1=“HealthCare”)
FILTER(?X2=“Full-Time”||
       ?X2=“Fulltime”)||
       ?X2=“Contract”)
FILTER(?X3=“^Euro”||
       ?X3=“^€”)
FILTER(?X4>=75000||
       ?X4<=120000)}
```

```
CONSTRUCT *
WHERE {
?Job :Category ?X1;
     :Role ?X2;
     :SalaryCurrency ?X3;
     :SalaryLower ?X4.
FILTER (?X1=“Health” ||
        ?X1=“BioSciences”)
FILTER(?X2=“Research\Academic
)
FILTER (?X3 = “UKP”)
FILTER (?X4 > 50000) }
```

```
SELECT ?Job
WHERE {
    ?Job :Location ?X1
    FILTER (?X1=“^UK” || ?X1=“^Belgium”)||?X1 = “^Germany”)
           || ?X1=“^Austria”)|| ?X1=“^Holland”))}
```

Figure 14. The SPARQL equivalent of Figure 13.

7. DISCUSSION AND FUTURE DIRECTIONS

This article proposed a language that allows people to query and mash up structured data without any prior knowledge about the schema, structure, vocabulary, or technical details of this data. Not only non-IT experts can use MashQL, but professionals can also use it to build advanced queries.

MashQL supports all constructs of the W3C standard SPARQL, except the "NAMED GRAPH" construct, which is introduced for advanced use, i.e. switching between different graphs within same query. To be close to user needs and intuition, we defined new constructs (e.g. OneOf, union "\", Without, Any, reverse "~", and others). The constructs are not directly supported in SPARQL, but emulated. We plan to include aggregation and grouping functions; especially as they are supported by Oracle's SPARQL.

Yet, MashQL does not support inferencing constructs (such as SubClass, or SubProperty), which are useful indeed for data fusion. As these constructs are expensive to compute (thus lead to bad interactivity of MashQL), we plan replace the Oracle's semantic technology that we are currently using as an RDF store, with an RDF index that we are developing, for speedy OWL inferencing.

We have downloaded most of the public RDF sources, on which our MashQL editor will be deployed online next month. Not only people will benefit from this, but we will also have the opportunity to better evaluate the usability of MashQL and its contribution to linking and fusing more data bottom-up.

Acknowledgement

We are indebted to Dr. George Pallis, Dr. Demetris Zeinalipour, and other colleagues for their valuable comments and feedback on the early drafts of this paper. This research is partially supported by the SEARCHiN project (FP6-042467, Marie Curie Actions).

REFERENCES

1 Athanasis N, Christophides V, Kotzinos D: Generating On the Fly Queries for the Semantic Web. ISWC (2004)

2 Abiteboul S, Duschkal O: Complexity of Answering Queries Using Materialized Views. ACM SIGACT-SIGMOD-SIGART. (1998)

3 Bizer C, Heath T, Berners-Lee T:Linked Data: Principles and State of the Art. WWW (2008)

4 Bloesch A, Halpin, T: Conceptual Queries using ConQuer–II. (1997)

5 Chong E, Das S, Eadon G, Srinivasan J: An efficient SQL-based RDF querying scheme. VLDB (2005)

6 Czejdo B, and Elmasri R, and Rusinkiewicz M, and Embley D: An algebraic language for graphical query formulation using an EER model. Computer Science conference. ACM. (1987)

7 Deng Y, Hung E, Subrahmanian VS: Maintaining RDF views. Tech. Rep CS-TR-4612 University of Maryland. 2004

8 Ennals R, Garofalakis M: MashMaker: mashups for the masses. SIGMOD Conference 2007:

9 Goldman R, Widom J: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. VLDB (1997)

10 Hofstede A, Proper H, and Weide T: Computer Supported Query Formulation in an Evolving Context. Australasian DB Conf. (1995)

11 http://demo.openlinksw.com/isparql (Feb. 2009)

12 Jarrar M, Dikaiakos: MashQL: A Query-by-Diagram Topping SPARQL. Proceedings of ONISW'08 workshop. (2008).

13 Jarrar M, Dikaiakos M: A query-by-diagram language (MashQL). Technical Article TAR200805. University of Cyprus, 2008. ttp://www.cs.ucy.ac.cy/~mjarrar/JD08.pdf

14 Kaufmann E, Bernstein A: How Useful Are Natural Language Interfaces to the Semantic Web for Casual End-Users. ISWC (2007)

15 Li Y, Yang H, Jagadish H: NaLIX: An interactive natural language interface for querying XML. SIGMOD (2005)

16 Popescu A, Etzioni O, Kautz H: Towards a theory of natural language interfaces to databases. 8th Con on Intelligent user interfaces. (2003)

17 Parent C, Spaccapietra S: About Complex Entities, Complex Objects and Object-Oriented Data Models. Info. System Concepts(1989)

18 http://rdfweb.org/people/damian/RDFAuthor (Jan. 2009)

19 Russell A, Smart R, Braines D, Shadbolt R.: NITELIGHT: A Graphical Tool for Semantic Query Construction. The Semantic Web User Interaction Workshop. (2008)

20 http://esw.w3.org/topic/SPARQL/Extensions? (Feb. 2009)

21 http://www.topquadrant.com/sparqlmotion (Feb. 2009)

22 Tummarello G, Polleres A, Morbidoni C: Who the FOAF knows Alice? A needed step toward Semantic Web Pipes. ISWC WS. (2007)

23 Zloof M: Query-by-Example:a Data Base Language. IBM Systems Journal, 16(4). (1977)

24 Zhuge Y, Garcia-Molina H, Hammer J, Widom J: View Maintenance in a Warehousing Environment. SIGMOD (1995)