Querying the Data Web

Technical Article, TAR200904

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

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

ABSTRACT

Given a source of structured data, how the casual user can query this data without prior knowledge about its schema. Some data sources are even schema-free or poorly-schematized. In this article, we present a graphical query formulation language, called MashQL. The main novelty of MashQL is that it allows people with limited IT-skills to query and explore one (or multiple) data sources without prior knowledge about the schema, structure, vocabulary, or any technical details of these sources. Users only use drop-down lists generated dynamically, as they interact with the query editor. Furthermore, to be more robust and cover most cases in practice (compared with related work), we even do not assume that a data source should have -an offline or inline- schema. To illustrate the query formulation power of MashQL, the Data Web is chosen as an application scenario in this paper. We also chose RDF as a data model and SPARQL as a backend query language. This is because RDF is the most primitive data model, and thus, MashQL can be similarly used for querying, e.g., databases and XML.

1. INTRODUCTION AND MOTIVATION

Allowing end-users to easily search and consume structured data is a known challenge, and started recently to receive a great attention from the Web 2.0 and the Data Web communities. In parallel to the continuous development of the hypertext web, we are witnessing a massive widespread of public structured data. Companies are competing not only on gathering structured content and making it public, but also on encouraging people to reuse and profit from this content. Many companies such as Google Base, Yahoo Local, Freebase, Upcoming, Flicker, eBay, Amazon, LinkedIn, and others, have made their content publicly accessible. In addition, companies have also started to widely adopt advanced web metadata standards. For example, Yahoo has officially announced [7] that if a web site points to or embeds RDF, it will be better presented in the search results. Several other models (such as RDFa, microformats, and RDF vocabularies) will also be supported by Yahoo. *MySpace* announced that they will adopt RDF for profile and data portability [30]. *Upcoming.org* is already publishing their content in microformats and RDFa. RDFa -a forthcoming W3C standard- is a new *simple* way of embedding RDF inside XHTML; so that machines can also understand the web.

This trend of structured 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. The results of a keyword-based query will not be precise or clean, because the query itself is still ambiguous although the underlying data is structured. To expose the massive amount of structured data to its full potential, people should be able to query this data easily and effectively. Formulating queries should be fast and should not require programming skills.

1.1 Challenges

The main challenge is that before formulating a query, one has to know the structure of the data and the attribute labels (i.e., the schema). End users are not expected to investigate "what is the schema" each time they search or filter information. In many cases, a data schema might be even dynamic, i.e., many kinds of items with different attributes are often being added and dropped (e.g., ebay). Other sources might be schema-free, or if it exists, the schema might be mixed up with the data (e.g., RDF). Allowing end users to query structured data flexibly is a challenge, especially when a query involves multiple sources.

Example: Figure 1 shows two RDF data sources[1] in the left-hand side. Suppose you want to "Retrieve Lara's articles after 2007". These sources do not only disagree on the labels of properties (e.g., Year versus PubYear), but also on data semantics. For example, while the rdf:Type property in the Example1 tells us that A1 and A2 are Articles, such knowledge does not exist in Example2; i.e., we do not know whether B1 and B2 are articles, books, or songs.

http://example1.com

```
:A1 rdf:Type bibo:Article
:A1 :Title "Data Web"
:A1 :Author "Tom Lara"
:A1 :Author "Bob Hacker"
:A1 :Year  2007
:A2 rdf:Type bibo:Article
:A2 :Title "Semantic Web"
:A2 :Author "Tom Lara"
:A2 :Year  2005
```

http://example2.com

```
:B1 :Title "Linked Data"
:B1 :Author "Lara T."
:B1 :PubYear  2008
:B1 :Publisher "Springer"
:B2 :Title "Data on the Web"
:B2 :Author "Abiteboul S."
```

SPARQL Query:

```
PREFIX S1:<http://example1.com>
PREFIX S2:<http://example2.com>
SELECT ?ArticleTitle
FROM <http://example1.com>
FROM <http://example2.com>
WHERE {{{?X S1:Title ?ArticleTitle} UNION
        {?X S2:Title ?ArticleTitle}}
 {?X S1:Author ?X1} UNION{?X S2:Author ?X1}
 {?X S1:Year ?X2} UNION {?X S2:PubYear ?X2}
 FILTER regex(?X1, "^Lara")
 FILTER (?X2 > 2007)}
```

Results:

ArticleTitle
Data Web
Linked Data

Figure 1. SPARQL query over two RDF data sources.

It is not necessary in RDF that data adheres to a certain schema or ontology, and if it does, it is inline with the data. Such data can be queried using SPARQL, the RDF query language [31]. The query in the right-hand side retrieves "the titles of the items that are written by Lara after 2007". Query conditions in SPARQL are called *triple-patterns*, and evaluated as pattern-filling [31], rather than truth-evaluation if compared with SQL. This is a robust technique for querying web data indeed. Not only schema-free data can be queried but also changes to data do not break queries. However,

[1] Data in RDF is represented as a set of triples <Subject, Predicate, Object>. Subjects and Predicates must be URIs, an Object must be either a URI or a Literal. In this way, an RDF sources can be seen as a directed label graph. We shall come back to this in later sections.

before writing a SPARQL query, one have to be fully-aware of the property labels and structures, which is difficult even for experts. In other words, unlike formulating SQL queries that start from small and offline schemes, formulating a SPARQL query requires one to manually investigate the data itself before querying it.

In short, **the problem statement**: How to allow people with limited IT-skills to query structured data, assuming that:

- *The user does not have to know the schema.* (1)
- *The data might be schema-free.* (2)
- *A query may involve multiple data sources.* (3)
- *The query language is sufficiently expressive.* (4)
  (i.e., not merely a single-purpose user interface)
- *The query language is intuitive for people with limited IT-skills.* (5)

Since some of these assumptions are subjective, they should be read rigidly. However, pinpointing the assumptions helps reading the article and remembering its contributions w.r.t to related works.

1.2 Our Contributions

We propose an interactive query formulation language, called MashQL. MashQL advances the state-of-the-art with a language that tackles all of the above assumptions. As we shall discuss in Section 2, there are similar query formulation methods considering some of these assumptions; however, up to our knowledge, none has tried to tackle all of them, which is indeed challenging. When comparing our approach with others, we do not consider the expressivity and intuitiveness assumptions (unless it is obvious), because of their subjective nature.

To be more concrete in illustrating the usefulness and the power of MashQL, we chose to focus on querying RDF, which is the most challenging data model; since it is the most primitive data representation model, and thus other models can be easily mapped into it [3]. Our contribution is subdivided into four major components:

- **Graphical query language**. This is the notational system and constructs that makes MashQL a formal, expressive, and yet intuitive query language. MashQL supports all constructs of SPARQL, allowing people to query and mash up multiple data sources at the same time, and redirect the results to be input to other queries. The design challenge of MashQL is the trade-off between its expressivity and intuitiveness.
- **Interactive query formulation algorithm**. This is the algorithm that the MashQL editor uses to allow users to interactively formulate a query. The challenging issue (and the novelty) of this algorithm is that it neither assumes the end user to know the data schema, nor the data itself to adhere to a certain schema (Assumption 1-2). As a user interacts with the editor, the editor queries the dataset (as a black-box) in the background and instantly offers the user the next choices.
- **Graph-Signature Index**. Because of assumption 2 (data is schema-free), the previous algorithm has to query the whole dataset at real-time. There is no offline schema of the data that can be used instead. Querying the whole dataset is challenge because queries may involve many joins, and thus, the interactivity of editor might be unacceptable in case of querying large and "deep" data graphs. Hence, we propose a new way of summarizing an RDF graph (called Graph-signature). Because the size of a graph-signature is typically much smaller than the original graph, queries become much faster, yielding an instant response-time by the MashQL editor.
- **Implementation and Evaluation**. Two different implementation scenarios of MashQL are presented in this paper: an online mashup editor, and a Firefox add-on mashup extension. The response-time of the MashQL editor is evaluated on querying two datasets (DBLP and DBPedia), and using both: our graph-signature index and Oracle's Semantic Technology. Our index shows that even a query involving long joins-path, can be answered instantly, regardless of the data size.

Paper organization: Section 2 positions our contributions w.r.t related works, and Section 3 gives a quick overview of MashQL. The query formulation algorithm is presented in Section 4, and the formalization of MashQL and its SPARQL translation rules are presented in Section 5 Section 6 presents the notion of query pipes. In Section 7 and Section 8 we present the graph-signature and the evaluation, respectively. Section 9 illustrates our implementation.

2. Related Work

Query formulation is the art of allowing people to easily query a data source (database, XML, or RDF). In the background, queries are translated into formal languages (SQL, XQuery, or SPARQL). This section reviews the main approaches to query formulation and how they relate to the novel contributions of MashQL.

Query-By-Form. This is the simplest approach to query data. All fields in a form are seen as query variables. However it is neither flexible nor expressive (fails with assumptions 2-4). For each query, a form needs to be developed; and changes to a query imply changes to its form.

Query-By-Example. Users formulate their queries as filling a table [39]. The names of the queried relations and fields are selected first; then users can enter their keywords. Although this approach is claimed to be easy for non-experts, it was not used by such users. This is mainly because users need to understand the schema among other issues (fails with assumptions 1, 2, and 5).

Conceptual Queries. As many databases are modeled at the conceptual level using EER or ORM diagrams, one can query these databases starting from these diagrams. Users can select certain concepts and relations from a given EER diagram, and their selection is automatically translated into SQL [14,33]. ORM approaches are different. RIDL [16] allows querying a database based on its ORM Schema but in a textual hybrid language. LISA-D [19] allows querying a database by converting its ORM schema into a list of facts, users can then drag-drop from this list. ConQuer [12] it starts from the logical schema and converts it into a list of concepts and a list of relations, based on its ORM diagram. Users can then formulate their queries by dragging and dropping from these lists. None of these languages was used in practice, since formulating a query starting from a conceptual diagram is still a difficult task for non-experts, (fails with assumptions 1,2,3, and relatively to 5, however we found [12] more intuitive).

Natural Language Queries allow people to write their queries as natural language sentences, and then translate these sentences into a formal language (e.g., SQL [32], XQuery [24]). Hence, people are not required to know the schema in advance. The main problem is that this approach is fundamentally bounded with the language

ambiguity and the "free mapping" between sentences and the data schemes (fails with assumptions 2,3, and relatively 4).

Visualize Triple Patterns. Several approaches in the semantic web community [2,6,34] propose to formulate a SPARQL query by visualizing its triple patterns as ellipses connected with arrows, so that one would need less programming skills to formulate a query (fail with assumptions 1 and 5).

Mashup Editors and Visual Scripting. Some mashup editors (e.g., Yahoo Pipes, Popfly, sMash) allow people to write query scripts inside a module, and visualize these modules and their inputs and outputs as boxes connected with lines. However, when a user needs to express a query over *structured data*, she has to use the formal language of that editor (e.g., YQL for Yahoo). Two approaches in the semantic web community [5,37] are inspired by this *visual scripting*. For example, [37] allows people to write their SPARQL queries (in a *textual form*) inside a box and link this box to another, in order to form a pipeline of queries. All of these visual scripting approaches are not comparable with MashQL, as they do not provide query formulation guide in any sense. They are included in this section, because *MashQL is also inspired by the way Yahoo Pipes visualizes query modules*. However, the *main* purpose of MashQL is not to visualize such boxes and links, but rather, to help formulating what is inside these boxes.

Interactive Queries. One of the most related approaches to MashQL is the Lorel query language [17]. Lorel was developed for querying XML graphically, and without assuming a user's knowledge about a schema (assumption 1). Lorel maps an XML document into a data graph, called EOM, which is close to RDF. The differences between MashQL and Lorel are: (First) Lorel assumes also the queried data to be schema-free (assumption 2), but it does not accurately handle this issue. Instead of querying the original data source, Lorel queries a summary of the data (called *dataguide*[28]). A *dataguide* is a small-size summary of all paths in an XML document. This summary is used to play the role of a schema. As we shall discuss this issue in details in section 7, a data guide groups many unrelated items together as they extrinsically use similar property labels. This was noted by the same authors in a later article [18]: "…*we have no way of knowing whether O is a publication, book, play, or song. Therefore, a DataGuide may group unrelated objects together*". The authors also noted that dataguides cannot be used for interactive queries as "*the worst case running time is exponential in the size of the database, and for a large database even linear running time would be too slow for an interactive session*". As the authors show the more the original data is graph-shaped (rather than tree-shaped as XML) the summary grows exponentially, and can be bigger than the original data graph. Notice that RDF is typically graph-shaped thus dataguide summaries are likely to be large. (Second) Lorel does not support querying multiple data sources (assumption 3) as MashQL does; and (third) its expressivity is very basic (assumption 4); however, MashQL supports path disjunctions and negations, variables, union, reverse properties, among many other operators.

Another relevant approach to MashQL is [24], which suggests a highly user interactive searching box. A user can write a keyword, the system then smartly and quickly suggests to auto-complete this keyword, based on a full index of the data and its schema. As such, the query becomes a set of schema-based annotated keywords. We found this approach intuitive as it is simple and does not assume any prior about the schema indeed (assumptions 1 and 5). However, unlike MashQL, the existence of a data schema is fundamental in this approach, and this is what makes it highly interactive. The problem with this approach also is that it cannot be play the role a query language (i.e., assumptions 2-3).

Being, at the sometime, expressive, intuitive, and highly interactive query language (over multiple, large, and schema-free data sources) is a very difficult challenge indeed. We refer to a recent usability study [23] about (which query formulation scenario the casual users prefer), which concluded that a query language should be close to natural language, graphically intuitive, and should not assume prior knowledge about the data.

3. A Quick Overview of MashQL

In this section we give a quick overview about MashQL. We start from the given the two RDF sources in Figure 1, and the SPARQL query to retrieve "Lara's articles after 2007". Figure 2 shows the same query but in MashQL. The first module specifies the query input, while the second module specifies the query body. The output of this query 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 MashQL queries; so to form a query pipe as will be shown in Figure 8.

Figure 2. The same SPARQL query in Figure 1, but in MashQL.

3.1 The general Intuition

Each MashQL query is seen as a tree. The root of this tree is called the *query subject,* which is the subject matter being inquired. A subject can be a particular instance (e.g., s2:A1 or s1:B1), an instance type (e.g., Article), or a variable label, such as '*Anything*', which is the default subject. Each branch of the tree is called a *restriction* and is used to restrict a certain property of the query subject. All branches (/restrictions) of the subject are conjunctive (i.e., an AND between them). Optional and negative restrictions are also supported; such restrictions are respectively prefixed with 'maybe' and 'without'. Branches can be expanded to allow sub trees, called *query paths*. In this case, the object of a property is seen as the subject of its sub query. As will be shown later the notion of query paths allows one to navigate through the underlying dataset and build complex queries. The symbol "☑" before a subject, property, or object indicates a projection (i.e., it will be produced in the results with its label as the column header).

When querying different sources, two properties (or two instances) are considered the same if an only if they have the same URI. To help end users not seeing cryptic URI, the query editor normalizes URIs, based on heuristics (see section **Error! Reference source not found.**) by simply detecting different namespaces of the same property labels and *optionally* combines them together. In case of different namespaces and property labels (e.g., S1:Year vs. S2:PubYear), the user can choose the union operator "\" to complain them.

Remark: As MashQL allows people to query multiple sources (especially through the union operator "\"); in a sense, MashQL can be used to integrate data. However, the goal of MashQL is data integration per se. Data integration is more complex, as it requires not only syntax but also semantic integration, which is not supported in MashQL. MashQL allows people to spot different labels of same properties (as they navigate data using the editor) and to manually combine them together.

Example 2: To illustrate the notion query paths, we extend the data given in Figure 1. We introduce new triples about the authors and their address (See Figure 3). Suppose you want to retrieve only the recent articles from Malta; i.e., retrieve the title of every article that has an author, this author has an address, this address as country called "Malta", and the article is published after 2007. This query path can be easily formed and easily understood in MashQL, as the query in Figure 3 shows.

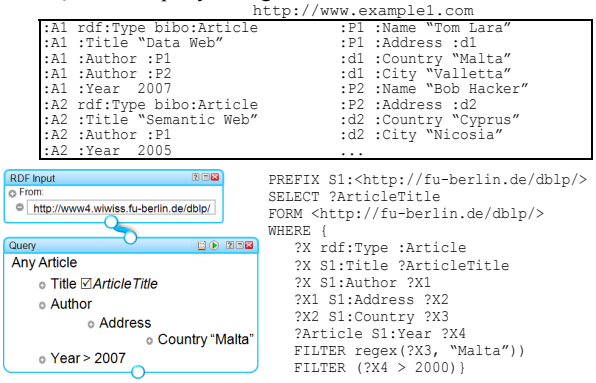


Figure 3. Query paths (/sub trees) in MashQL

Formulating a query is an interactive process; users only select from drop-down lists. While interacting with the query editor, the editor instantly queries the dataset to dynamically generate these lists. First, the editor queries a given dataset (as a black-box) to find the main concepts/instances, from which the query subject can be selected (e.g. `Article`). The editor then finds the possible properties for this subject (e.g. `Title`, `Author`, `Year`). The user can select a property (e.g. `Year`) and restrict it using a filtering function (e.g. `MoreThan`) and a filter value (e.g. `2007`). If a user selects to expand a property (e.g. `Author`), the editor will find the set of properties that this 'Author' may have (e.g. `Address`, `Name`), and so on. In this way, users can query and navigate a data source without any prior knowledge about it. The algorithm that the editor uses to generate "the next list" is formalized and explained in Section 4. To achieve extremely fast user-interaction, in case of large and deep graphs, a graph signature index has been developed and evaluated (Section 7 and 8).

3.2 Intuitiveness

The trade-off between expressivity and simplicity in MashQL is achieved by making *technical variables and namespaces to be implicit*, and specially through *the tree structure of MashQL* queries, which is close to the intuition people use in their natural language communication. For example, the query path shown in Figure 3 means, retrieve the article that has an Author x_1, and x_1 has an address x_2, and x_2 has a country x_4, and x_4 equals "Malta". Because the query is represented as a tree, these variables are implicit for end users. Furthermore, suppose you would like to ask; "Give me the list of all stores that sell parts of the iPhone mobile, and that are located in Rome"; or, "Which cinemas are located in Lyon, offer a movie called 'Fahrenheit' and will be played between 20:00 and 23:00". Notice that apart from some terms (such as: give me the list of all, which, who, that are), all of these inquiries can be directly converted into MashQL queries.

4. QUERY FORMULATION ALGORITHM

In this section, we propose a query formulation algorithm, by which the complexity and the responsibility of understanding a data source (even it is schema-free) are moved from the user to the query editor. The algorithm is formally described in the steps below; Figure 4 is a walk-through demo of this algorithm.

Step 0: *Specifies the dataset D* in the Input module. D can be one or a merge[2] of multiple data graphs.

Step 1: *Select the query subject S*, where $S \in S_T \cup S_I \cup V$. In other words, after specifying the dataset users can select ***S*** from a drop-down list that contains, either: (i) S_T: the set of the subject-types in the dataset, such as `Article` as in Figure 3; or (ii) S_I: the union of all subject and object identifiers in the dataset. Users can also choose (iii) not to select from the list but introduce their own *subject label*. In this case, the subject is seen as a variable and displayed in *italic,* which means anything ($S \in V$). These three options are formalized respectively as the following, in *both* relational algebra and SPARQL:

(1) $S \in S_T : \boldsymbol{\pi}_O(\boldsymbol{\sigma}_{P=':Type'}(D))$
(1') `O1:{(?S1 <:Type> ?O1)}`
(2) $S \in S_I : \boldsymbol{\pi}_S(D) \cup \boldsymbol{\pi}_O(\boldsymbol{\sigma}_{O \in URI}(D))$
(2') `S1:{(?S1 ?P1 ?O1)} UNION O1:{(?S1 ?P1 ?O1).Filter isURI(?O1)}`
(3) $S \in V$
Repeat step 2-3 (until the user stops)

Step 2: *Select a property P*. Depending on the chosen subject in step 1, a list of the possible properties for this subject is dynamically generated. There are four possibilities: (i) if ($S \in S_T$), such as `Article`, then the list will be set of all properties that the instances of the subject-type have may {`Title`, `Author`, `Year`}. (ii) if ($S \in S_I$), such as `A1`, the list will the set of all properties that this instance may have. (iii) If the subject is a variable ($S \in V$), the list will be the set of *all* properties in the dataset. (iv) users can also choose the property to be a variable by introducing their own label. See the formalization[3] of these four options below. Furthermore, users can choose to make the selected property required, optional, or unbound. As discussed in the next section (see Figure 6), if a property is prefixed with "maybe" this property is considered optional, if it is prefixed with "without" it is considered unbound, and if it is not prefixed then it is required.

(4) $(S \in S_T) \rightarrow P \in \boldsymbol{\pi}_{P2}(\boldsymbol{\sigma}_{P1=':Type' \wedge O1=Subject}(D) \bowtie_{S1=S2} \boldsymbol{\sigma}(D))$
(4') `P2:{(?S1 <:Type> <S>)(?S2 ?P2 ?O2)}`
(5) $(S \in S_I) \rightarrow P \in \boldsymbol{\pi}_P(\boldsymbol{\sigma}_{S=Subject}(D))$
(5') `P1:{(<S> ?P1 ?O1)}`
(6) $(S \in V) \rightarrow P \in \boldsymbol{\pi}_P(\boldsymbol{\sigma}(D))$
(6') `P1:{(?S1 ?P1 ?O1)}`
(7) $P \in V$

2 Merging RDF graphs is straightforward as specified in the W3C standard [31], All triples are put together. Two nodes or two edges are considered exactly the same if and only if they have the same labels (i.e., URI).

3 To avoid confusion with variable and attribute names, when self-joins are used, we assume that the attributes of a source D are named as (S_1, P_1, O_1), (S_2, P_2,O_2)… (S_n, P_n,O_n).

Step 3: *Add a filter on P.* There are three types of restrictions: object filter, object identifier, or query path. (i) If a user wants to add an object filter on the previously selected property, a set of functions will be offered (e.g., `Equals`, `Contains`, `Doesn't contain`, `OneOf`, `Between`, `MoreThan`, `Not`, etc.). (ii) If a user wants to add an object identifier as a restriction, a list of the possible objects will be generated. For example, if a user previously chose `Article` as a subject, and `Author` as a property, the set of object identifiers would be {`A1`, `A2`}, given the dataset in Figure 3. The formalizations given below (8-13) specify what the list of objects identifier may contain, taking into account the previously chosen subject and property. Furthermore, (iii) users can also chose to expand the property to declare a query path on it, such as `Author` in Figure 3. In this case, the value of the property `Author`, which is a variable here, will be considered as the subject of this sub-query. The possible properties of this subject (in the 2nd level) will be determined as described in step 1, taking into account the previous selections. The general case formalization of an *n*-level property and *n*-level object are presented below (14-17), in both cases, where the root subject is a type and is an instance.

(8) $(S \in S_I) \wedge (P \in V) \rightarrow O \in \pi_{O1}(\sigma_{S1=S \wedge O1 \in URI}(D))$
(8') `O1:{(<S> ?P1 ?O1) Filter isURI(?O1)}`
(9) $(S \in S_I) \wedge (P \notin V) \rightarrow O \in \pi_{O1}(\sigma_{S1=S \wedge P1=P \wedge O1 \in URI}(D))$
(9') `O1:{(<S> <P> ?O1) Filter isURI(?O1)}`
(10) $(S \in S_T) \wedge (P \in V) \rightarrow O \in \pi_{O2}(\sigma_{P1=':Type' \wedge O1=S}(D) \bowtie_{S1=S2} \sigma(D))$
(10') `O1:{(?S1 <:Type> <S>)(?S1 ?P2 ?O2)}`
(11) $(S \in S_T) \wedge (P \notin V) \rightarrow O \in \pi_{O2}(\sigma_{P1='rdf:Type' \wedge O1=S}(D) \bowtie_{S1=S2} \sigma_{P2=P}(D))$
(11') `O:{(?S <rdf:Type> <S>)(?S <P> ?O)}`
(12) $(S \in V) \wedge (P \in V) \rightarrow O \in \pi_O(\sigma(D))$
(12') `O1:{(?S1 ?P1 ?O1)}`
(13) $(S \in V) \wedge (P \notin V) \rightarrow O \in \pi_O(\sigma_{P=P}(D))$
(13') `O1:{(?S1 <P> ?O1)}`

General Cases

The n-level paths properties and objects, in case $(S \in S_T)$
(14) $P \in \pi_{Pn}(\sigma_{P1=':Type' \wedge O1=S}(D) \bowtie_{S1=S2} (\sigma_{C2}(D) \bowtie_{O2=S3} (\sigma_{C3}(D) \ldots \bowtie_{On-1=Sn} (\sigma_{Cn}(D)))))$
(14') `Pn:{(?S1 <Type> O)(?S1 P2 O2)(O2 P3 O3) … (On-1 ?Pn ?On)}`
(15) $O \in \pi_{On}(\sigma_{P1=':Type' \wedge O1=S}(D) \bowtie_{S1=S2} (\sigma_{C2}(D) \bowtie_{O2=S3} (\sigma_{C3}(D) \ldots \bowtie_{On-1=Sn} (\sigma_{Cn}(D)))))$
(15') `On:{(?S1 <Type> O)(?S1 P2 O2)(O2 P3 O3)…(On-1 Pn ?On)}`

The n-level paths properties and objects, in case $(S \in S_I)$
(16) $P \in \pi_{Pn}(\sigma_{C1}(D) \bowtie_{O1=S2} (\sigma_{C2}(D) \bowtie_{O2=S3} (\sigma_{C3}(D) \ldots \bowtie_{On-1=Sn} (\sigma_{Cn}(D)))))$
(16') `Pn:{(S1 P1 O1)(O1 P2 O2 … (On-1 ?Pn ?On)} \\Subject ∈ SI`
(17) $O \in \pi_{On}(\sigma_{C1}(D) \bowtie_{O1=S2} (\sigma_{C2}(D) \bowtie_{O2=S3} (\sigma_{C3}(D) \ldots \bowtie_{On-1=Sn} (\sigma_{Cn}(D)))))$
(17') `On:{(S1 P1 O1)(O1 P2 O2)(O2 P3 O3)…(On-1 Pn ?On)}`

Step 4: The symbol ☑ can be used before subject, property, or object variables to indicate that this variable will be returned in the results (i.e., projection).

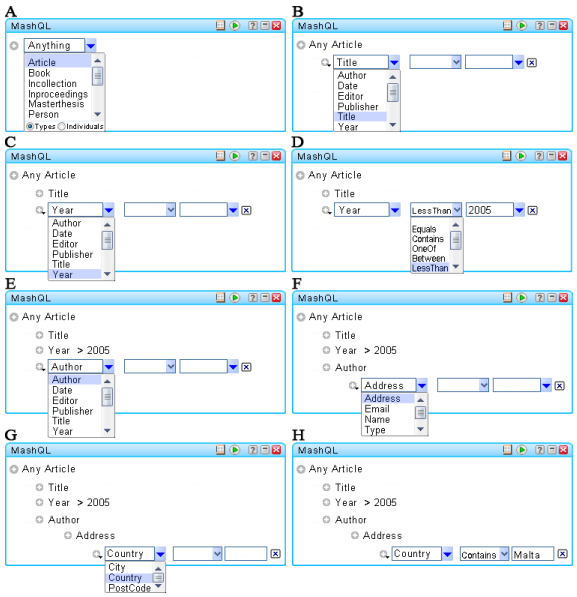


Figure 4. A Query Formulation Demo.

5. THE DEFINITION OF MASHQL AND ITS TRANSLATION INTO SPARQL

5.1 Data Model

MashQL assumes the queried dataset is structured as (or mapped into) a directed labeled graph, similar to but not necessarily the exact RDF syntax. The minimum definition of a dataset D is a set of data triples <Subject, Predicate, Object>. A subject and a predicate can only be a unique identifier *I* (a URL or a key). The value of an object can be a unique identifier *I* or a literal *L*.

Def.1 (Dataset): *A dataset D is a set of triples forming a directed labeled graph, each triple t is formed as <S, P, O>, where* $S \in I$, $P \in I$, *and* $O \in I \cup L$.

The only difference between this definition and the RDF model is that we allow an identifier to be any form of a key (i.e. weaker than a URI). Allowing this, would simplify the use of MashQL for querying databases. Relational databases (or XML) can be directly converted to this *primitive* data structure. In Figure 5 we show a simple example of mapping (or viewing) a database to a directed labeled graph. The primary key of a table is seen as a subject, a column label as a predicate, and the data-entry in that column as an object. Foreign keys represent relationships between data elements across tables. Mapping from database and XML into RDF is a mature topic and is entering a standardization phase [3].

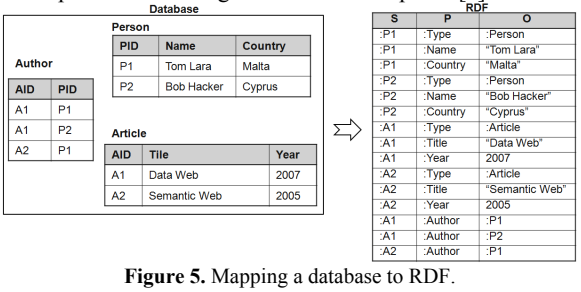


Figure 5. Mapping a database to RDF.

Furthermore, we assume that each object literal must have a datatype. If an object value does not have an explicit datatype, it can be implicitly assumed by taking advantage of XML and RDF

conventions: in XML, the syntax for literals is a String, enclosed in double or single quotes; Integers are written directly without quotes; Booleans are written as true or false; and so on. In RDF and XML Schema, stating a datatype explicitly is done using namespaces, such as: `"1"^^xsd:integer, "2004-12-06"^^xsd:date.`

Def.2 (Typed Literals): *Every object literal must have a datatype D: If $O \in L$ then $O \in D$.*

Object literals may also have a language tag L_t (such as "En", "Gr", "It") to indicate to which language the object value belongs. In the RDF best practice, language tags are expressed using `@` followed by a language tag, such as `"Person"@En, "Ατομο"@Gr.`

Def.3 (Language Tags): *An object literal ($O \in L$) may have a language tag L_t.*

5.2 The Formal Definition of MashQL

As a MashQL query itself cannot be executed, but, its SPARQL translation that is execute, the semantics of MashQL follows the semantics of SPARQL. Table 1 presents the formal definition of the MashQL constructs, and Table 2 presents their SPARQL interpretation.

Similar to SPARQL, when evaluating a query *Q(S)*, only the triples that satisfy all restrictions are retrieved, such that: First, if a restriction is not prefixed, (R:=<*empty, P, O_f*>), see Def.6, the truth-evaluation of the restriction is considered true if the subject, the predicate, and the object-filter are matched (see the first two restrictions in Figure 6). This case is mapped into a normal graph pattern in SPARQL (see rule-3). Second, if a restriction is prefixed with "*maybe*" (R:=<*maybe, P, O_f*>), its truth-evaluation is considered always true (see the 3rd restriction in Figure 6). This case is mapped into an optional graph pattern in SPARQL (see rule 4). Third, if a restriction is prefixed with "*without*" (R:=<*without, P, Of*>), it is truth-evaluation is considered true if the subject *S* and the predicate *P* do not appear together in a triple (see the last restriction in Figure 6). Notice that there is no such a construct in SPARQL, but in MashQL, it means that the object *O* should not be bound (see rule 5).

Example. The query in Figure 6 means: retrieve everything (call this thing as a *Song*) that: has a title, for the artist Shakera, possibly has an Album, and does not have a Copyright. In other words, when evaluating this query, we retrieve all triples that have same subject and: 1) with a predicate Title, 2) with a predicate Artist and the object identifier is Shakera, 3) maybe with a predicate Album, and 4) should not have the predicate Copyright.


```
SELECT ?SongTitle, AlbumTitle
WHERE {
{?X :Title ?SongTitle.
 ?X :Artist :Shakera
 Optional{?X :Album ?AlbumTitle}
 Optional{?X :Copyrihgt ?X1}
 Filter FILTER (!Bound(?X1)).
```

Figure 6. A MashQL query and its mapping into SPARQL.

As defined in Def.7, MashQL supports 9 forms of object filters: Equals, Contains, MoreThan, LessThan, Between, OneOf, Not, and query paths. Not all of these functions have a direct support in SPARQL but they are emulated somehow (see rules 6-13). MashQL also supports a union operator between objects, properties, subjects, and queries (see Def.8, and rules 14-17). In addition, to allow people formulate queries at the type level, the construct "Any" before a subject or object means that the instances of this subject/object will be retrieved instead of the subject/object itself. As shown in rules 18-19, types in RDF are indicated by the property "`rdf:type`". Furthermore, since RDF is a *directed* graph, it is very helpful for a user to explore this graph backward. This is supported in my MashQL by the Reverse construct (see Def.9 and rule 20).

Table 1. The formal definition of MashQL

Def. 4 (Query): *A Query Q with a subject S, denoted by Q(S), is a set of restrictions on S. $Q(S) := R_1 \wedge \ldots \wedge R_n$.*

Def. 5 (Subject): *A subject $S \in (I \cup V)$, I is an identifier, V is a variable.*

Def. 6 (Restriction): *A restriction $R :=$ <R_x , P, O_f>, R_x is an restriction prefix $R_x \in$ {empty, maybe, without}; P is a predicate ($P \in I \cup V$); O_f is an object filter.*

Def.7 (Object Filter): *An object filter $O_f :=$ <O, f>, O is an object, f is a filtering function. f can have one of the following nine forms:*

1. *$O_f :=$ <O>, where O is an object, $O \in V \cup I$.* This object filter does not add any restriction on the object value as shown in Figure 6.
2. *$O_f :=$ <O, Equals(X, T, L_t)>, where X can be a variable or a constant, T is a datatype, and L_t is a language tag.* See the restriction (Country "Malta") in Figure 3. See the mapping rule-6 to SPARQL.
3. *$O_f :=$ <O, Contains(X, T, L_t)>, where O is an object variable, X is a regex literal, T is a data type, and L_t is a language. O* should be equal to regex(*X*). See the restriction (Author "^Lara") in Figure 2. see rule-7.
4. *$O_f :=$ <O, MoreThan(X, T)>, where O is an object variable, X is a variable or a constant, T is a datatype.*
5. *$O_f :=$ <O, LessThan(X, T)>, where O is an object variable, X is a variable or a constant, T is a datatype identifier.*
6. *$O_f :=$ <O, Between(X, Y, T)>, where X and Y are variables or constants, T is a datatype identifier.*
7. *$O_f :=$ <O, OneOf(V)>, where O is an object variable, and V is a set of values {v_1, ... , v_n}, v_i is a variable or constant.*
8. *$O_f :=$ <O, Not(f)>, where f is one of the functions defined above.* This filter extends all of the above functions with simple negation.
9. *$O_f :=$ <O, $Q_i(O)$>, where O is an object ($O \in V \cup I$), and $Q_i(O)$ is a sub-query with O being the query subject.* The restrictions defined in the sub-query $Q_i(O)$ should be satisfied as well.

Def.8 (Union): A union can be declared between objects, predicates, subjects and/or queries, in the following forms:

1. *O_n = <O_1\$O_2$\…\$O_n$>, to indicate unions between objects, $O_i \in I$.*
2. *P_n = <P_1\$P_2$\…\$P_n$>, to indicate unions between predicates, $P_i \in I$.*
3. *S_n = <S_1\$S_2$\…\$S_n$>, to indicate unions between subjects, where $S_i \in I$.*
4. *Q_n = <Q_1\$Q_2$\ . . . \$Q_n$>, to indicate unions between queries*,

Def.8 (Types): *A subject ($S \in I$) or an object ($O \in I$) can be prefixed with "Any" to mean the instances of this subject/object type.*

Def.9 (Reverse): *<~P> indicates the reverse of the predicate P. Let R_1 be a restriction on S such that <S P O>, and R_2 be <O ~P S>, R_1 and R_2 have the same meaning.*

Table 2. MashQL-To-SPARQL mapping rules

The following rules map the MashQL constructs into SPARQL:

Rule-1: The symbol ☑ before a variable means that it will be returned in the results; i.e., included in the SELECT part of in SPARQL. If the output of the query is input to another, use CONSTRUCT *

Rule-2: if a subject, predicate, or object in a MashQL query is *italicized*: it is seen as a SPARQL variable, i.e. prefixed with "?".

Rule-3: If *S* is a subject and *R* = < , *P, Of*>, the mapping is: `{S P O}.`

Rule-4: If *S* is a subject and *R* = <*maybe, P, Of*>, the mapping is: `{OPTIONAL{S P O}}.`

Rule-5: If *S* is a subject and *R* = < *without, P, Of*>, the mapping is: `{S P O. FILTER (!bound(?O))}.`

Rule 6. If O_f = <*O, Equals(X, T, Lt)*>:
Append the mapping with: `FILTER(?O = X)`
If *T* ≠ *Null*: Append the mapping with: `FILTER(datatype(?O)=T)`
If L_t ≠ *Null*: Append the mapping with: `FILTER(lang(?O) = Lt)`

Rule 7. If O_f = *Contains(X, T, Lt)*>:
Append the mapping with: `FILTER regex(?O, X)`
If *T* ≠ *Null*: Append the mapping with: `FILTER(datatype(?O)=T)`
If L_t ≠ *Null*: Append the mapping with: `FILTER(lang(?O) = Lt)`

Rule 8. If O_f = <*O, MoreThan(X, T)*>:
Append the mapping with: `FILTER(?O > X)`
If *T* ≠ *Null*: Append the mapping with: `FILTER(datatype(?O=T)`

Rule 9. If O_f = <*O, LessThan(X, T)*>:
Append the mapping with: `FILTER(?O < X)`
If *T* ≠ *Null*: Append the mapping with: `FILTER(datatype(?O=T)`

Rule 10. If O_f = <*O, Between(X, Y, T)*>:
Append the mapping with: `FILTER(?O >=X)&& FILTER(?O<=Y)`
If *T* ≠ *Null*: Append the mapping with: `FILTER(datatype(?O)=T)`

Rule 11. If O_f = <*O, OneOf (V)*>: Append the mapping with:
`{FILTER(?O = V1)|| . . . || FILTER(?O = Vn)}`
If V_i is a regex-ed literal, the i^{th} filter above should be replaced with:
`FILTER Regex(?O, Vi)`

Rule 12. If O_f = <*O, Not(f)*>: *f* filter is generated as above, but with a negation.

Rule 13. If O_f = <*O, $Q_i(O)$*>:Repeat all mapping rules to generate $Q_i(O)$.

Rule 14. Given O_n , If *n >1* and $O_i \in I$: The mapping in rules 3-4 will be:
`{{S P :O1} UNION . . . UNION {S P :On}}`

Rule 15. Given P_n , If *n >1* and $P_i \in I$: The mapping in rules 3-4 will be:
`{{S :P1 O} UNION . . . UNION {S :Pn O}}`

Rule 16. Given S_n , If *n >1* and $S_i \in I$: Regenerate the query *n* times, each time with S_i as a root, and with a UNION between the queries.

Rule 17. Given Q_n , If *n >1* : Add UNION between the n queries.

Rule 18. If a subject *S* is prefixed with "any": `{?S rdf:type :S}`

Rule 19. If an object *O* is prefixed with "any": `{?O rdf:type :O}`

Rule 20. If *S* is a subject and *R* = <~*P, O*>, the mapping is: `{O P S}.`

6. THE NOTION OF QUERY PIPES

A simple scenario of using MashQL is to place it as a query interface topping a relational or a graph database; however, in an open world scenario other challenges might be faced. This section overviews these challenges (only from a query formulation viewpoint) and introduces the notion of *query pipes*.

Suppose one creates a mashup by querying Upcoming.org to retrieve all events happening in Paris (Q_1); another person creates another mashup based on Q_1 to only retrieve the scientific events (Q_2); a third person mashes up the results of Q_2 with the events retrieved from the ACM Conferences (Q_3); and so on. We call queries that connect to each other in this way as *pipe*. This notion is important in open worlds as it enables people to collaborate and build on each others' efforts. Allowing people to formulate query pipes is not merely a visualization of links between query modules, but when compiling a pipe (i.e., translating its queries into SPARQL, or SQL), some query formulation should be considered.

6.1 Translating MashQL into SELECT and CONSTRUCT statements.

(First) Translating MashQL into SELECT statements in SPARQL is not enough, because the SELECT statement produces the results in a tabular form. To allow queries to input each other, the results of a query should be formed as a graph. This is not important when using SQL, because the input and the output of a query have the same tabular form. In SPARQL, the CONSTRUCT statement produces a graph, but then one needs to manually specify how this graph should be produced. To overcome this and allow a MashQL query to produce the results in both tabular and graph forms, 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 [3]. 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, the SELECT statement will be generated (see Figure 10).

In short, in case MashQL is used to formulate a pipe over a relational database, all queries in this pipe are translated into SQL SELECT statements. However, in case of formulating pipe over an RDF graph, each query will be translated into either SELECT or CONSTRUCT statements, depending on the pipe structure.

6.2 Cashing and Materialization

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[4]; 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. This can be easy done in case of querying relational databases, by translating MashQL into Views and Materialized Views statements. However, as this is not supported by SPARQL, a similar framework is defined below.

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 the pipes consistency, two updating strategies are used: Query auto-refresh and Pipe auto-refresh. MashQL maintains for each base or derived source *D* a

[4] A query with medium size complexity over a large dataset takes one or few seconds [13].

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 (e.g., [10, 38]) an efficient refreshing strategies is the *incremental updates* strategy, 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 [15], because RDF data and queries are developed and maintained autonomously by different people.

6.1 Query Pipes Use Case

Bob has a PhD in bioinformatics. He is looking for a fulltime, well-paid, and research-oriented job in some countries. Instead of visiting many job portals and each time trying many keywords and filters, Bob used MashQL to query these portals according to his preferences (Figure 8). First, he visited Google and Jobs and performed a keyword search on each site (bioinformatics OR "systems biology" OR e-health). He copied the links of the retrieved results (Figure 7) from Google and Jobs into the Input modules[5], and created two queries on the results. The third query integrates the results and filters them based on location.

http//:base.google.com/jobs/rdf	http//:www.jobs.ac.uk/rdf
<:job2> :Title "Project Manager E-Health"	<:1> dc:Title "Senior Research Manager"
<:job2> :Type "Contract"	<:1> :Category "Health"
<:job2> :Salary 60000	<:1> :Location "London, UK"
<:job2> :Currency "Euro"	<:1> : MinimumSalary 55000
<:job2> :Location "Paris, France"	<:1> : MaximumSalary 85000
<:job2> :Employer "UniLife"	<:1> :SalaryCurrency UKP
<:job2> :JobIndustry "Insurance"	<:1> :Role "Research/Academic"
<:job3> :Title "Genome Annotation"	<:2> dc:Title "PhD scholarship"
<:job3> :Type "Full-Time"	<:2> :Category "BioSciences"
<:job3> :Salary 77000	<:2> :Location "Manchester, UK "
<:job3> :Currency "€"	<:2> :MinimumSalary 21000
<:job3> :Location "Gent, Belgium"	<:2> :MaximumSalary 21000
<:job3> :Employer "BioCom"	<:2> :SalaryCurrency UKP
<:job3> :JobIndustry "Healthcare"	<:2> :Role "Research/Academic"

Figure 7. Sample of RDF data about jobs.

[5] We assume Google and Jobs.ac.uk render their search results in RDFa i.e. RDF triples are embedded in HTML. Otherwise, Bob need to use a third party's service (e.g. Dapper) to extract triples from HTML.

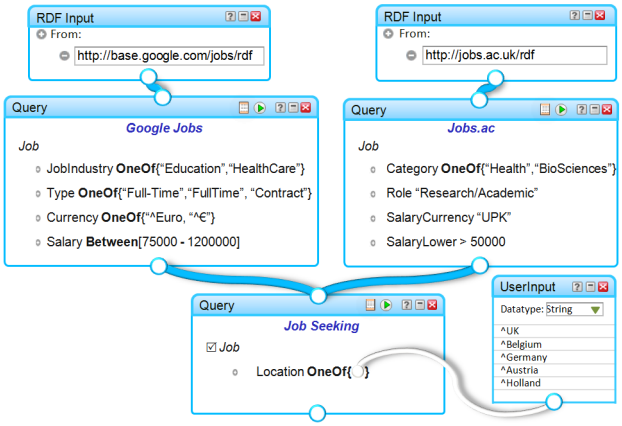


Figure 8. Bob's MashQL Queries.

Job
http//:base.google.com/jobs/rdf/job3
http/:www.jobs.ac.uk/rdf/1

Figure 9. The results of piped queries in Figure 8

```
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 10. The SPARQL translation of the MashQL queries in Figure 8.

7. The Graph-Signature Index

Recall that when formulating a MashQL query, the editor queries the data in the background to generate the list of next choices. In such a user-interaction setting the response-time be small, preferably less than 100ms [25]. However, such queries might be slow since a graph is typically stored in one relational table, RDF(S,P,O). Suppose the graph in Figure 11 is stored in such a table, and you want to know the "list of properties of the authors of A1": {?P: (A1 Author ?O1) (?O1 ?P ?O2)}; or "the countries of the affiliation of the authors of A1": {?O3: (A1 Author ?O1) (?O1 Affiliation ?O2) (?O2 Country ?O3)}. Such queries might be inefficient as they involve several self joins. MashQL's background queries are a special case of graph\SPARQL queries; they are only linear join-paths. A background query can be 1) what is the set of properties $\boldsymbol{P_n}$ of a given subject S, of the form $\{\mathbf{P_n}: (S\ P_1\ ?O_1)(?O_1\ P_2\ ?O_2)\ldots\ (O_{n-1}\ P\ O_n)\}$, where $n \geq 1$; 2) what is the set of objects $\boldsymbol{O_n}$ of a subject S, through a chain of properties, of the form $\{\mathbf{O_n}: (S\ P_1\ ?O_1)(?O_1\ P_2\ ?O_2)\ldots\ (?O_{n-1}\ P\ O_n)\}$, where $n \geq 1$. Executing a query of level n, implies self joining the RDF table n-1 times.

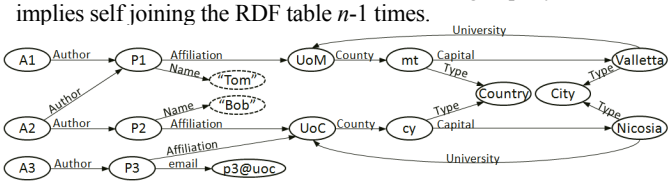


Figure 11. An RDF data graph.

Several approaches have been proposed to index RDF graphs for fast queries. Among these, Oracle suggested in [13] to build a subject-property matrix materialized join views on the RDF table, such that, all direct and nested properties for a group of subjects is materialized. This approach (called Semantic Technology) has been released as part of Oracle 10g and 11g. Another approach (called C-Store [8]) claimed a better performance over Oracle by

partitioning the RDF table vertically, into *n* two-column tables, where *n* is the number of unique properties in the data. In other words, a table (S,O) is created for each property. A more recent approach (called RDF3X [29]) claimed a better performance over both, by simply building many B^+-Tree indexes, and "*careful optimization of complex join queries*" [29]. We have tried to use these approaches as a backend RDF index for MashQL, and we found that non can reach an acceptable user-interaction (<100ms response-time) with large data graphs. As explained earlier MashQL's background queries are a special case, as they form only join-path queries; rather than, e.g., star-like queries, which is claimed to be a strength of [29].

The problem -of joins-path queries- has been studied also by the XML community, and the notion of *structural summaries* has been proposed. The idea is to summarize all paths in a given XML document, so that, a long joins-path can be optimized when executing an XQuery. The size of a structural summary is typically very small, compared to the original XML document, because it contains only the paths (i.e., the structure) between XML elements; such as {Author.Affiliation.Name, Author.Affiliation.country.capital, etc.}. One of the most known structural summary approaches is called *DataGuide*[28]. This approach was later used by the Lorel [18] query formulation language (see Section 2). A DataGuide contains all and only the possible paths in an XML document. The problem with this approach is that unrelated entities are grouped together [18]. As also stated by its proposers a DataGuide "*can tell you that a link labeled "Book" can only be followed by links labeled "Author"... but cannot tell you if every link labeled "Book" is followed by a link labeled "Author*" [28]. That is, when asking "what are the properties of the Authors of A3", the answer would be {Affiliation, email Name}. Notice that Name is not a property of A3. The reason is that all possible properties followed by Author are combined together. Another idea by the authors has been proposed, called Strong DataGuide, which guarantees that unrelated entities will not be grouped together. However, the problem is that the size of a DataGuide can grows exponantionaly and can be bigger than the original data in case this data is graph-shaped. In other words, Strong DataGuides behaves better in case of XML, as it is tree-shaped. Among many other approaches, F&B index [] proposed to group nodes in a graph if and only iff they reachable by all incoming and outgoing paths, i.e. indistinguishable by any forward and backward path. This approach also guarantees (unlike DataGuides) that the worst case for the size of a summary (upper-bound) will be always less than or equals the original graph. The problem with this approach is that the size of the summary is often large. That is, it better behaves in case of XML tree-shaped data.

In the following we propose a new summarizing approach suitable for RDF graphs, which we call *graph-signature index*. The idea is similar to XML structural summaries as it generates a small-size summary of an RDF graph; so that, certain queries (in particular, MashQL's formulation queries) can be answered from this small summary, much faster than querying the whole original graph. Our RDF graph-signature is fundamentally different from XML structural summaries. Not only that it holds information about all subjects in the graph, but more importantly, because RDF is not the same as XML. "*RDF triples form a graph rather than a collection of trees*" [29].

The intuition of the graph-signature is to have a graph of categories that summarizes the original RDF graph. Nodes are grouped if they have the same set of outgoing paths, i.e., indistinguishable by any backward path. In other words, Similar subjects in an RDF graph are grouped into categories, such that, a category is the set of all subjects that have exactly the same property labels, and the objects of each of their properties belong to same categories. Each category is given a unique identifier. We also define a *category signature* as the set of properties that some subjects share.

Def.1. (**Category**) Given two subjects S_1 and S_2, we say that they have the same category C_i, if and only if:

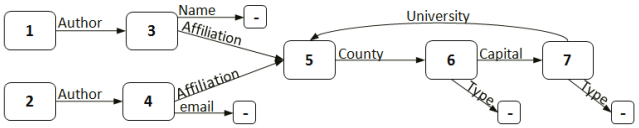
1. *They share the same property labels*: There exist (S_1 P_1 O)… (S_1 P_m O), m>0, and (S_2 P_1 O) … (S_2 P_n O), n>0, such that, the distinct set of properties of S_1 $\{P_1..,P_m\}$ equals the distinct set of properties of $S_1\{P_1,..,P_n\}$.
2. *The objects of each of their properties belong to the same categories*: for each property P_i of S_1 there is O_1, and P_i of S_2 there is O_2, such that, O_1 and O_2 belong to a same category C_j. Notice that C_i and C_j, can be same or different categories.
3. If O is a literal or it does not belong to the set of all subjects in the dataset, its category is *null.*

Def.2 (**Category Signature**) A Category signature is a set of triples of the form <SC, P, OC>, where SC is a category of some subjects, P is a property label, and OC is a category of some objects.

Def.3 (**Graph Signature**) is the set of all category signatures.

Although Def.1 is recursive, for a brevity, however generating the graph signature is not. See the algorithm in Figure 14.

Example. The data graph in Figure 11 is summarized by the graph shown in Figure 12, and alternatively in the table GraphSignature. The SubjectCat table indexes each subject and its category. Notice, for example, that P1 and P2 are assigned the same category 3, because they share the same set of property labels {Affiliation, Name}. However, P3 is assigned another category 4, as its properties are not the same as P1 and P3. It has an email and does not have a name. A1 and A2 have the same properties but they are assigned different categories, because (see Def.1.2) the objects of their properties do not have the same category. That is, the category of A1.Author is 3, while the category of A3.Auhtor is 4. UoM and UoC, and mt and cy, and Valletta and Nicosia are assigned the categories 5, 6, 7 respectively; as they satisfy all requirements in Def.1.



SubjectCat

S	C
A1	1
A2	2
P1	3
P2	3
P3	4
UoM	5
UoC	5
mt	6
cy	6
Valletta	7
Nicosia	7

Graph Signature

SC	Predicate	OC
1	Author	3
2	Author	4
3	Name	-
3	Affiliation	5
4	email	-
4	Affiliation	5
5	Country	6
6	Capital	7
6	Type	-
7	Type	-
7	University	5

Figure 12. Graph-Signature index.

Note that adding more instances of the same category into a dataset, does not increase the size of its graph signature. In other words, the size of the graph signature will be smaller than the size of the original graph. As will be shown in the next section, the DBLP dataset (a graph with 9M Million triples) generated a graph signature of 1.2K triples. The DBpedia dataset (a graph 32 Million triples), generated a graph signature of 156K triples. As we shall evaluate in the next section, the importance of a graph signature is that querying it, is cheaper than querying the original

graph. Because of its small size (thus fits in a small memory), self joining is cheaper than self joining the original graph. Furthermore, a graph signature holds information about all subjects in the dataset, not only paths, compared with structural summaries. For example, the answer of "what are the properties of the Authors of A1" is {Affiliation, name}, and "the properties of the Authors of A3" is {Affiliation, email}. However, a DataGuide's answer would be {Affiliation,email, name}, because all links following Author are grouped together.

To gain further efficiency in querying and storage, we replace all string labels in RDF with unique integers, as described in [13]. In addition, to support the "rdfs:SameAs" property. For example, if we have a triple (A3 rdfs:SameAs A4), both A3 and A4 will be given the same identifier, before generating the graph signature.

Furthermore, not all RDF sources are schema-free. For example, we found that every subject in the DBLP dataset is described by the property Type. In DBPedia, not all but the majority of the subjects are typed. Hence, it is important to make use of these types when formulating a query. For example, because mt and cy are typed with Country it might be better to ask "Give me the Capital of any Country". To support this as a build-in feature in MashQL, we extend the SubjectCat table to include Types (see Figure 13).

Type	S	C
	A1	1
	A2	2
	P1	3
	P2	3
	P3	4
	UoM	5
	UoC	5
Country	mt	6
Country	cy	6
City	Valletta	7
City	Nicosia	7

Figure 13. Typed subjects are included in Graph-signature index.

In this way, MashQL can be used to formulate queries starting from a certain instance as discussed in the beginning of the section; or starting from types, if they exist. As also discussed earlier, a user can start a query by a variable. Recall that in this case the variable label is displayed in *italic*. The default query subject is the variable *Anything*, see Section 4. For example, "what are the properties of authors of anything" {?P2: (?AnyThing Author ?O1)(?O1 P2 ?O2)}. Moreover, MashQL allows predicates to be variables. For example, "what are the properties of the properties of the properties of A1", {P4: (A1 ?P1 ?O1)(?O1 P2 ?O2) (?O2 P3 ?O3) (?O3 P4 ?O4)}. As we will discuss in the next section, such queries are expensive in Oracle, but became cheap using the graph-signature. The idea of allowing such queries in MashQL is to allow one to, for example, find the relations between two entities, at a certain degree of relationships.

To conclude, in this section we have proposed the notion of graph signature to summarize a given RDF graph. As the size of such summaries tends to be much smaller than the original data graph, joins-path queries (*for the purposes of MashQL*) become cheaper than joining the whole data graph. In the next section, we evaluate MashQL's background queries using the graph signature and compare it with using Oracle's Semantic Technology.

Remark: To avoid any misconception, we would like to emphasize that our graph signature is designed only for query formulation purposes. In other words, the graph signature is indexed and contains only the information that is needed to formulate a MashQL query. Hence, we do not claim the graph signature to be a *general purpose RDF index,* to e.g., answer any SPARQL query as the Oracle's Semantic Technology index does. Exending the graph signature for query optimization is a future work, which is very promising indeed. We basically build two separate graph signatures for each data graph, one is to index all incoming baths and one to index all outgoing baths. The intersection of the answers of the tow signatures is the smallest superset of the target answer.

An algorithm to create a Graph-Signature

Input: *RDF(S,P,O)*
Output: *SubjectSignature(S,Cat)*, *CatSignature*(*Cat*, *Predicate*, *NextCat*)

P = The set of all distinct *predicates* in *RDF*;
Create Table *SPredicates* (*S*,P_1,P_2,…P*n*); \\ *n* is the number of predicates is P.
For *i = 1…n* {
  Pred = lookup the next *predicate* from *P*;
  Update *SPredicates* set P_i = *Pred* where *S* in (Select *S* from *RDF* where *P* = *Pred*);}
Create Table *SCat* (*Cat*,P_1,P_2,…P*n*);
Insert into *SCat*(P_1…P_n) Select P_1…P_n from *SPredicates* group by P_1…P_n;
Assign a sequential *id* to *Cat* for each row in *SCat*;
Alter Table *SPredicates* add column *Cat*;
Assign a hash key for similar (P_1,…,P*n*) in *both SPredicates and SCat.*
\\A hash can be generated for every concatenation(P_1,…,P*n*), this becomes the category id.

Alter Table *SPredicates* Drop columns (P_1,…,P*n*); \\ The result is *SPredicates*(*S*,*Cat*)
Alter Table *RDF* add column *CatId*, *NextCat*;
Assign *Cat* for each *S* in *RDF*. \\ Can be done by left joining *RDF* and *SPredicates* on S=S;
Assign *NextCat* id for each *O* in *RDF*. \\ left joining *RDF* and *SPredicates* on O=S;

Insert into *CatSignature* Select *Cat,P,NextCat* from *RDF* group by *Cat,P,NextCat*;
Insert into *SubjectCat*(*S,Cat*) Select *S,Cat* from *RDF* group by *S,Cat.*
Alter Table *RDF* drop column (*Cat,NextCat*); Drop Table *SPredicates*. Drop Table *Predicates*;

Figure 14. An algorithm to create a graph-signature index.

8. Evaluation

We present two types of evaluations. 1) We evaluate the scalability of the graph-signature index. 2) We evaluate the time-cost of formulating a MashQL query using this index, and compare it with using the Oracle's Semantic Technology.

8.1 Datasets and Experiment Settings

Our evaluation is based on two public datasets: A) DBLP and B) DBPedia. The DBLP (700MB size) is a graph of 9 million edges. We partitioned this graph into three parts: A9 is the whole DBLP; A4 is 4 million triples from A9; and A2 is 2 million. No sorting is used to partition the data. We only used "Create Table A4 as select * from A9 where rownum<4000001". The original A9 is not sorted either. Figure 16 shows some statistics about these partitions. The DBPedia (6.7 GB) is graph of 32 million edges. Similarly, DBPedia is partitioned into 4 parts (see Figure 16). DBPedia is the RDF version of the whole Wikipedia. Each part of the two datasets is loaded into a separate relational table -of the form RDF(S,P,O). The DBMS used to store these tables is Oracle 11g, which was installed on a server with: 2GHz dual CPU, 2 GB RAM, 500GB HHD, 32-bit Unix OS.

Number of	(A)DBLP			(B)DBPedia			
	A9	A4	A2	B32	B16	B8	B4
UniqueTriples	9M	4M	2M	32M	16M	8M	4M
UniqueSubjects	1.1M	1M	0.8M	9.4M	6M	4M	2.6M
UniquePredicates	28	27	26	35	35	34	34
UniqueObjects	2.4	1.2	0.7M	16M	8.7M	4.7M	2.5
Data Size	700MB	350MB	170MB	6.7GB	3.1GB	1.4GB	550MB

Figure 15. Statistics about the experimental data.

8.2 Scalability Evaluation

A graph-signature is built on each partition of the datasets. Figure 16 shows some statistics. As one can see, the time cost to build a graph signature is liner w.r.t data size. For example, B4 (4M triples) cost 285 seconds, the time is almost doubled when the data size is doubled (B8 costs 528sec, B16 costs 1177, etc).

Number of	(A)DBLP			(B)DBPedia			
	A9	A4	A2	B32	B16	B8	B4
Indexing Time (Sec)	429	216	162	2378	1177	528	285
Unique Categories	136	6K	3K	6K	32K	16K	6K
Triples in GraphSignature	1.2K	53K	23K	165K	486K	185K	56K

Figure 16. Statistics about the graph-signatures of all parts of the dataset.

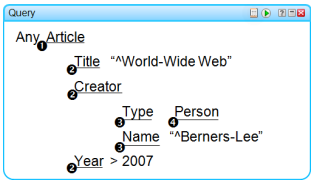
What is more scalable, is the behavior of the index w.r.t the number of the triples . For example, the whole DBPedia B32

(32M triples) is summarized into 156k triples; this number tends to increase when the data is smaller, 486K for B16. Similarly, the whole instances in DBLP A9 (9M triples) are summarized by 1.2K triples, but half of this data A4 generated 53K triples. This means, it is likely that the more data we have, the more similarities (i.e., categories) are found, hence, the smaller the graph-signature. In fact, the index reflects the homogeneity of a graph. For example, because DBLP is indeed well-structured (i.e., most instances have similar properties), it is summarized only by 136 categories. However, the DBPedia (which is automatically extracted from the Wikipedia, with too much noise; in fact, large parts of it, is meaningless), it generated more categories. Anyway, for both datasets, the generated graph-signatures fit in a small memory, which yields fast queries as the next evaluation will show.

8.3 Response-Time Evaluation

This section evaluates the response-time of the MashQL editor's user interaction. In other words, we are *not* interested to evaluate the execution of a MashQL query itself, as this is not the purpose of the graph-signature; but rather, the execution of the queries that the editor performs in the background to generate the "next" drop-down list (see Section 4). In the following we presented three MashQL queries. We identify the set of background queries, and execute them using: (1) the graph-signature index, which we implemented in Oracle 11g, and (2) Oracle's Semantic Technology, which is the native RDF index in Oracle 11g.

Experiment 1: To formulate the query in Figure 17 on DBLP, the user first selects the query subject from a list. The query produces this list is annotated by ❶. The user then selects a property of this subject from a list. The query produces this list is annotated by ❷; and so on. These queries are executed on each partition of the DBLP, using both: the graph-signature (GS) and Oracle's Semantic Technology. The cost[6] (in seconds) is shown in Figure 18.



❶ O:(?S Type ?O)
❷ P:(?S Type Article)(?S ?P ?O1)
❸ P:(?S Type Article) (?S Creator ?O1) (?O1 ?P ?O2)
❹ O:(?S Type Article) (?S Creator ?O1)(?O1 Type ?O)

Figure 17. 4 queries are needed to formulate this MashQL query.

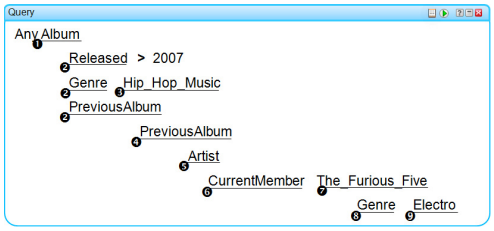
Query	(A9) 9 M triples		(A4) 4 M triples		(A2) 2 M triples	
	GS	Oracle	GS	Oracle	GS	Oracle
Q_1	0.003	0.005	0.003	0.004	0.003	0.003
Q_2	0.001	0.136	0.001	0.148	0.001	0.108
Q_3	0.001	0.187	0.001	0.846	0.001	0.671
Q_4	0.001	1.208	0.001	0.835	0.001	0.650

Figure 18. Time cost (in ms) of background queries.

As shown by this experiment, the time cost for each query remains within few milliseconds using our index, regardless of the data size. This is because the size of the index is small, compared with the Oracle's Semantic Technology that scans the whole dataset.

Experiment 2: Here we show a similar evaluation on DBPedia, but with longer queries (see Figure 19). Each query is executed on all partitions, and the time-cost is shown in Figure 20.

[6] To avoid the I/O dominance, we did not include GROUB-BY and ORDER-BY, and only the top 10000 rows are retrieved.



❶ O:(?S Type ?O)
❷ P:(?S Type Album)(?S ?P ?O1)
❸ O:(?S Type Article)(?S Genre ?O)
❹ P:(?S Type Album)(?S PreviousAlbum ?O1)(?O1 ?P ? O2)
❺ P:(?S Type Album)(?S PreviousAlbum ?O1)(?O1 PreviousAlbum? O2)(?O2 ?P ?O3)
❻ P:(?S Type Album)(?S PreviousAlbum ?O1)(?O1 PreviousAlbum? O2) (?O2 Artist ?O3)(?O3 ?P ?O4)
❼ O:(?S Type Album)(?S PreviousAlbum ?O1) (?O1 PreviousAlbum? O2) (?O2 Artist ?O3)(?O3 CurrentMember ?O4)
❽ P:(?S Type Album)(?S PreviousAlbum ?O1) (?O1 PreviousAlbum? O2) (?O2 Artist ?O3)(?O3 CurrentMember The_Furious_Five)(The_Furious_Five ?P ?O5)
❾ O:(?S Type Album)(?S PreviousAlbum ?O1) (?O1 PreviousAlbum? O2) (?O2 Artist ?O3)(?O3 CurrentMember The_Furious_Five)(The_Furious_Five Genre ?O5)

Figure 19. 9 queries are needed to formulate this MashQL query.

Query	(B32) 32 M		(B16) 16 M		(B8) 8 M		(B4) 4 M	
	GS	Oracle	GS	Oracle	GS	Oracle	GS	Oracle
Q_1	0.003		0.003		0.003		0.003	
Q_2	0.002		0.002		0.002		0.002	
Q_3	0.005		0.004		0.003		0.003	
Q_4	0.005		0.004		0.004		0.004	
Q_5	0.005		0.004		0.004		0.004	
Q_6	0.005		0.005		0.005		0.005	
Q_7	0.007		0.007		0.007		0.007	
Q_8	0.005		0.005		0.005		0.005	
Q_9	0.007		0.007		0.007		0.006	

Figure 20. Time cost of the background queries in Figure **19**.

Similar to the previous experiment, this experiment also shows that the time cost, for all queries remains very small indeed, although the dataset is larger, more heterogeneous, and the queries involve longer join-path expressions.

Experiment 3(Extreme): This experiment might not be faced in practice; but its goal is to exposes the limits of both, our index and Oracle's index. Figure 21 shows a query where all properties are *variables*. It means, what are the properties of the properties of … (at 9 degree) of properties of any Album. After selecting Album as the query subject, and then move to select from the list of its properties, the user decides to make the property as a variable, at each level. The query editor, at each level, generates the list of the possible properties depending on the previous selections. For example, at the 2nd level, the editor's query, ❷: `P2:(?S Type Album)(?S ?RelatedTo1 ?O1)(?O1 ?P2 ?O2);` at the 3rd level, ❸: `P3:(?S Type ?Album)(?S ?RelatedTo1 ?O1)(?O1 ?RelatedT02 ?O2)(?O2 ?P3 ?O4);` and so on. Notice that executing such queries is very expensive as the *whole* index must be scanned and joined with itself i_{-1} times, at level i.

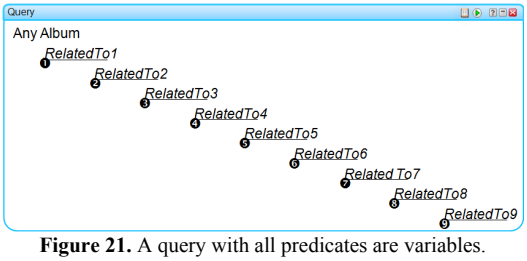


Figure 21. A query with all predicates are variables.

Query	(B32) 32 M		(B16) 16 M		(B8) 8 M		(B4) 4 M	
	GS	Oracle	GS	Oracle	GS	Oracle	GS	Oracle
Q_1	0.001	0.027	0.001		0.001		0.001	
Q_2	0.006	17.450	0.026		0.010		0.005	
Q_3	0.011	49.656	0.048		0.010		0.010	

Q_4	0.017	5348.7	0.087	-	0.022	-	0.011	-
Q_5	0.032	-	0.135	-	0.036	-	0.011	-
Q_6	0.047	-	0.185	-	0.055	-	0.016	-
Q_7	0.061	-	0.234	-	0.076	-	0.023	-
Q_8	0.077	-	0.265	-	0.098	-	0.027	-
Q_9	0.092	-	0.280	-	0.110	-	0.034	-

Figure 22. A query involving many background joins.

This is indeed the worst-case scenario for both indexes. As shown in Figure 22, Oracle's Semantic Technology did not respond after the 4rd level. On the other side, although the execution time using our index increases at each level, but the important thing is that this increase remains fairly acceptable, for such type of extreme queries. The reason for our index to be faster, is that the graph signature fits in a small memory, even with some magnitudes of self joins. However, Oracle's Semantic Technology performs the self joins on the whole dataset, which is too large. In other words, our index joins only the graph signature, which is 165K edges, however Oracle's joins the whole data graph, which is 32M edges.

To conclude, as shown by these three experiments, because the size of the graph-signature index is small, long join-path queries can be executed very fast. This speed enables the MashQL editor to perform its background queries instantly, regardless of the dataset.

9. IMPLEMENTATION AND DISCUSSION

MashQL can be used in different scenarios to query data. An obvious scenario is to place it on top of a relational or a graph database. For example as many offline or online databases need to queried by non-IT experts in a flexible manner, such as , about health, scientific, traffic data (e.g., see [35]). MashQL can be used as a graphical query interface of such databases. In the following we overview two different usage scenarios that we have implemented: an online server-side mashup editor, and a browser-side Firefox plug-in.

9.1 Online Mashup Editor

We have developed an online mashup editor, where MashQL is used as a data mashup language (see Figure 23 and its system model in Figure 22). The functionality of this tool comprises: i) the MashQL language components; ii) the user-interface; iii) a state-machine dispatching the "background queries" in order to support query formulation during the interactive exploration of RDF datasets; iv) a module that translates a formulated MashQL query into SPARQL and submits this for execution, or debugging; the formulated MashQL query is serialized in XML; v) a module that retrieves, merges, and presents the results of the submitted SPARQL query. Users can query and mash up web 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, and our graph-signature index is used to achieve instant response-time user interaction. When a user specifies a data source(s) as input, it is bulk-loaded and indexed. MashQL queries are also translated into Oracle's SPARQL[7]. While interacting with the editor to formulate a query, the editor performs some background queries through AJAX. Users can store and publish their queries so that others can discover, reuse, or clone. Each published query is given a URL. Calling this URL means executing this query and getting its results back. In this way, the output of a query is seen as a concrete RDF source. Queries (that require high execution cost) and primer input sources (that require high downloading and bulk-loading costs) are materialized and refreshed periodically.

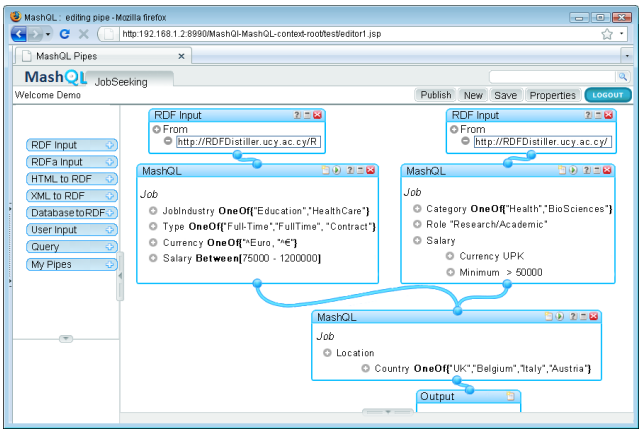


Figure 23. **MashQL Editor Screenshot.**

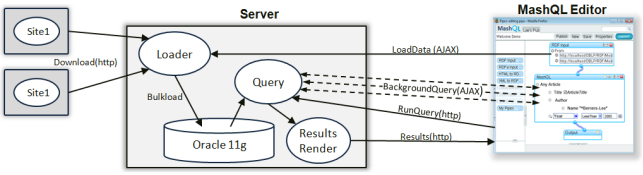


Figure 24. System Model

As one may notice, MashQL's GUI follows the style Yahoo Pipes visualizes feed mashups, by using the Yahoo open-source Java-Script libraries. Our choice of following this style is to illustrate that MashQL can be used to query and mash up the Data Web as simple as filtering and pipes web feeds. In this way is a data mashup is a query over multiple sources. It is worth noting also that the examples of this article cannot be built using Yahoo pipes. Yahoo allows a limited support of XML mashups, but this is neither graphical nor intuitive; as one have to write complex scripts in YQL, the Yahoo Pipes' query language.

9.2 Firefox add-on MashQL editor

We have also developed a Firefox add-on in order to allow developing mashups at the client side (See Figure 25). The left side shows the MashQL query module. When creating a mashup, the visited pages -in the browser tabs- are automatically selected as input sources to this query. Users can exclude some of them, and/or add local files. The results are rendered into a new tab, and can be saved locally. Users can also save queries and choose their periodic refreshments. Unlike the online mashup editor, where data sources are loaded into and queried from a database, this client-side editor uses the Jena library to parse, cache, and query data directly from the memory of the client. Thus, it is faster and cheaper solution, but, the amount of the processed data is limited by the memory. The goal of this implementation is to allow querying and fusing websites that embed RDFa. In this way, the browser is used as a Web composer rather than only a navigator. Because not many pages support RDFa yet, users currently need to use a third-party service (such as triplr.org, buzzword.org.uk, or wandora.org) to distil the RDF triples from HTML. When

[7] Oracle has extended SQL with a table-function to allow SPARQL-like queries [13]. For simplicity, we call this extension as "Oracle's SPARQL". Notice that in this paper we only present the translation of MashQL into the W3C standard SPARQL, MashQL to Oracle's SPARQL is not presented here for space limitation.

many pages will include RDF, the goal of this plug-in is to allow one, for example, to compose his publication list from Google Scholar, DBLP, ACM, and CiteSeer; or, filter all video lectures given by Berners-Lee from YouTube, VedioLectures, and iTunes.

Figure 25. MashQL Firefox add-on (an early screenshot).

9.3 Implementation Issues

URI Normalization: As RDF data may contain unwieldy URIs, MashQL queries might be inelegant. Thus, the editor normalizes URIs and displays the normalization instead; for example, Type instead of http://www.w3.org/1999/02/22-rdf-syntax-ns#type. In addition, if one moves over Type, its URI is displayed as a 'tip'. Internally, the editor uses only the long URIs. In case of different URIs leading to the same normalization, we add a gray prefix to distinguish them (e.g., 1:Type, 2:Type). The normalization is based on a repository that we built for the common namespaces (e.g., rdf, rdfs, WOL, FOAF). In case a URI does not belong to these namespaces, the editor uses heuristics. For example, takes the last part after '#'. If '#' does not exist, then the part after '/'. The result should be at least 3 characters and start with a letter, otherwise we take the last two parts of the URL, and so on. We have evaluated this on many datasets and found it covering the extreme majority of cases. However, there is no guarantee to always produce elegant normalization.

Verbalization: To further improve the elegancy of MashQL, we use a *verbalize*/*edit* modes. When a user moves the mouse over a restriction, it gets the edit mode and all other restrictions get the verbalize mode. That is, all boxes and lists are made invisible, but their content is verbalized and displayed instead (See Figure 4). This makes the queries readability closer to natural language, and guides users to validate whether what they see is what they intended.

Scalable lists: In case of querying large datasets, the usual drop-down list becomes un-scalable. We have developed a scalable and friendly list that supports search, auto-complete, and sorting based on Rank and Asc/Desc. If Rank is selected, we order items/nodes based on how many nodes points to them. This knowledge is pre-computed, from the Graph Signature. Our list supports also scalable scrolling. The first 50 results are displayed first, but one can scroll to go to the next, arbitrarily middle, or last 50. Each time the editor sends an AJAX query to fetch only those 50. Furthermore, our list allows the user to select either Instances (i.e., any URI in the dataset) or Types of instances (i.e., instances that have the predicate rdf:Type if exits, such as Author, Person, University). See the lists in Figure 4. When a user selects the option Type, the MashQL editor generates different background queries, i.e., at the type level (See the specification of these queries in section 4).

Acknowledgment We are indebted to many people who gave us many valuable comments and suggestion, including, George Pallis, Yannis Ioannidis, Demetris Zeinalipour, Rizos Sakellariou, Vasilis A. Vassalos. This research is partially supported by the SEARCHiN project (FP6-042467, Marie Curie Actions).

References

1 http:// agraph.franz.com/allegrograph (Mar. 2009)
2 http://demo.openlinksw.com/isparql (Mar. 2009)
3 http://www.w3.org/2005/Incubator/rdb2rdf/ (Mar. 2009)
4 http://esw.w3.org/topic/SPARQL/Extensions? (Mar. 2009)
5 http://www.topquadrant.com/sparqlmotion (Mar. 2009)
6 http://rdfweb.org/people/damian/RDFAuthor (Mar. 2009)
7 http://www.ysearchblog.com/archives/000527.html (Mar.08)
8 Abadi D, Marcus A, Madden S, Hollenbach K: Scalable semantic web data management using vertical partitioning. VLDB, 2007.
9 Athanasis N, Christophides V, Kotzinos D: Generating On the Fly Queries for the Semantic Web. ISWC, 2004.
10 Abiteboul S, Duschkal O: Complexity of Answering Queries Using Materialized Views. ACM SIGACT-SIGMOD-SIGART. (1998)
11 Blackwell F, Britton C, Cox A, Green T, Gurr C, Kadoda G, Kutar M, Loomes M, Nehaniv C, Petre M, Roast C, Roe C, Wong A, Young R: Cognitive dimensions of notations: Design tools for cognitive technology. 4th Conf. on Cognitive Technology, Springer. (2001)
12 Bloesch A, Halpin, T: Conceptual Queries using ConQuer–II. ER, 1997
13 Chong E, Das S, Eadon G, Srinivasan J: An efficient SQL-based RDF querying scheme. VLDB'05, Springer. 2005
14 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
15 Deng Y, Hung E, Subrahmanian VS: Maintaining RDF views. Tech. Rep CS-TR-4612 University of Maryland. 2004
16 De Troyer O, Meersman R, Verlinden P: RIDL on the CRIS Case: A Workbench for NIAM. Proc. of IFIP WG 8.1 Working. (1988)
17 Goldman R, Widom J: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. VLDB, 1997
18 Goldman R, Widom J: Interactive Query and Search in Semistructured Databases. WebDB, 1998
19 Hofstede A, Proper H, and Weide T: Computer Supported Query Formulation in an Evolving Context. Australasian DB Conf. (1995)
20 Jarrar M, and Dikaiakos M: MashQL: A Query-by-Diagram Language. ONISW'08, part of the ACM CiKM conference. ACM. (2008).
21 Jarrar M, and Dikaiakos M.: MashQL: A Query-By-Diagram Language for Data Mashups. Tech. Article (TAR200805). University of Cyprus.
22 Jayapandian M, Jagadish H: Automated Creation of a Forms based Database Query Interface. VLDB, 2008
23 Kaufmann E, Bernstein A: How Useful Are Natural Language Interfaces to the Semantic Web for Casual End-Users. ISWC, 2007.
24 Li Y, Yang H, Jagadish H: NaLIX: an interactive natural language interface for querying XML. SIGMOD, 2005
25 Miller R: Response time in man-computer conversational transactions. AFIPS 1968
26 Moro M,Vagena Z, Tsotras V: Evaluating Structural Summaries as Access Methods for XML. WWW 2006
27 Nandi A, Jagadish H: Assisted querying using instant-response interfaces. SIGMOD, 2007.
28 Nestorov S, Ullman J, Wiener J, Chawathe S: Concise Representations of Semistructured Hierarchical Data. ICDE 1997.
29 Neumann T, Weikum G: RDF3X: RISC style engine for RDF. VLDB 2008
30 O'Donoghue, J.: MySpace joins the 'semantic' web. The Web User online magazine. May 9, 2008
31 Prud'hommeaux E, Seaborne A: SPARQL Query Language for RDF. 2008
32 Popescu A, Etzioni O, Kautz H: Towards a theory of natural language interfaces to databases. 8th Con on Intelligent user interfaces. 2003
33 Parent C, Spaccapietra S: About Complex Entities, Complex Objects and Object-Oriented Data Models. Information System Concepts, 1989

34 Russell A, Smart R, Braines D, Shadbolt R.: NITELIGHT: A Graphical Tool for Semantic Query Construction. SWUI Workshop. 2008.

35 Schomburg I, Chang A, Ebeling C, Gremse M, Heldt C, Huhn G, Schomburg D: BRENDA, The enzyme database: updates and major new developments. Nucleic acids research. (2004)

36 Spyns P, Oberle D, Volz R, Zheng J, Jarrar M, Sure Y, Studer R, Meersman R: Semantic Web Community Portal. PAKM, 2002.

37 Tummarello G, Polleres A, Morbidoni C: Who the FOAF knows Alice? A needed step toward Semantic Web Pipes. ISWC Workshops. 2007

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

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