# A Query Formulation Language for the Data Web

Mustafa Jarrar, Marios D. Dikaiakos

**Abstract**— We present a query formulation language (called MashQL) in order to easily query and fuse structured data on the web. The main novelty of MashQL is that it allows people with limited IT-skills to explore and query one (or multiple) data sources without prior knowledge about the schema, structure, vocabulary, or any technical details of these sources. More importantly, to be robust and cover most cases in practice, we do not assume that a data source should have -an offline or inline- schema. This poses several language-design and performance complexities that we fundamentally tackle. To illustrate the query formulation power of MashQL, and without loss of generality, we chose the Data Web scenario. We also chose querying RDF, as it is the most primitive data model; hence, MashQL can be similarly used for querying relational databases and XML. We present two implementations of MashQL, an online mashup editor, and a Firefox add-on. The former illustrates how MashQL can be used to query and mash up the Data Web as simple as filtering and piping web feeds; and the Firefox add-on illustrates using the browser as a web composer rather than only a navigator. To end, we evaluate MashQL on querying two datasets, DBLP and DBPedia, and show that our indexing techniques allow instant user-interaction.

**Index Terms**— Query Formulation, Semantic Web, Data Web, RDF, SPARQL, Indexing Methods, Query Optimization, Mashup

## 1. Introduction and Motivation

Allowing end-users to easily search and consume structured data is a known challenge that receives recently a great attention from the Web 2.0 and the Data Web communities. The rapid growth of structured data on the Web has created a high demand for making this content more reusable and consumable. 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, and LinkedIn have made their content publicly accessible through APIs. In addition, companies have also started to widely adopt web metadata standards. For example, Yahoo started to support websites embedding RDF and microformats, by better presenting them in the search results; MySpace also started to adopt RDF for profile and data portability; Google, Upcoming, Slideshare, Digg, the Whitehouse, and many others started to publish their content in RDFa, a forthcoming W3C standard for embedding RDF inside webpages so that content can be better understood, searched, and filtered.

This trend of structured data on the Web (Data Web) is shifting the focus of Web technologies towards new paradigms of *structured-data retrieval*. Traditional search engines cannot serve such data as 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 on the Web 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. Other sources might be schema-free, or if it exists, the schema might be inline 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 sources[1], Example1.com and Example2.com. Suppose a Web user wants to retrieve "Lara's articles after 2007" from both sites. These sources do not only disagree on property labels (e.g., Year and PubYear), but also on data semantics. For example, while the rdf:Type in Example1 tells us that A1 and A2 are Articles, we do not know whether B1 and B2 in Example2 are articles, books, or songs.

It is not necessary in RDF that data adheres to a certain schema or ontology. RDF data is queried using SPARQL [42]. 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 [41], rather than truth- evaluation if compared with SQL. This is a robust way for querying schema-free data, as changes to data do not cause queries to break; however, it poses hard query formulation challenges. Before writing a query, one has to be fully-aware of the property labels and data structures. Unlike formulating SQL,

• *M. Jarrar is the Birzeit Universit, Ramallah, Palestine. E-mail:mjarrar@birzeit.edu*
• *M. D. Dikaiakos is with the University of Cyprus.20537, Nicosia, Cyprus. E-mail:,mdd@cs.ucy.ac*

[1] RDF represents data as a directed labeled graph. A graph is a set of triples of the form <Subject, Predicate, Object>. Subjects and Predicates must be URIs, an Object can be either a URI or a Literal.

```
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.

query requires one to manually investigate the *data* itself before querying it. This issue becomes challenging in the case of large datasets; and even more complex when querying multiple sources, as predicates have to be explicitly union-ed (See Figure 1).

As discussed in section 2, allowing people to easily query and consume structured data is a known challenge in different areas. However, in an open environment, as the Data Web, for a query formulation language to be practically sound, it should address the assumptions below:

**Position Statement**: How to allow people with limited IT-skills to query structured data, assuming that:

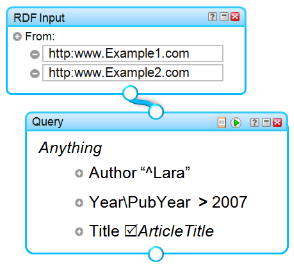
- *The user does not have to know the schema.* (1)
- *The data might be schema-free.* (2)
- *A query may involve multiple data sources.* (3)
- *The query method is sufficiently expressive.* (4)
  *(i.e., not merely a single-purpose user interface)*

## 1.2 Overview of Contributions

We propose an interactive query formulation language, called *MashQL*. The novelty of MashQL (compared with related work) is that it considers all of the above assumptions together. Being a language -not merely an interface and, at the same time, assuming data to be schema-free is one of the key challenges addressed in the context of MashQL design and development. Without loss of generality, this article focuses on the Data Web scenario. We regard the Web as a database, where each data source is seen as table. In this view, a *data mashup* becomes a query involving multiple data sources. To illustrate the power of MashQL we chose to focus on querying RDF, which is the most primitive data model, hence, other models -as XML and relational databases - can be easily mapped into it [4].

We give a bird's-eye view of MashQL in Figure 2 which shows the same query as in Figure 1 written in MashQL. The first module specifies the query input, the second module specifies the query body, and the output is piped into a third module (not shown here) that renders the results into HTML or XML, or as RDF input to other queries.

Each MashQL query is seen as a tree; the root is called the *query subject*. Each branch is a *restriction* on a property of the subject. Branches can be expanded to allow sub trees (Figure 4), called *query paths*, which allows one to navigate through the underlying dataset and build complex queries. Formulating a query is an interactive process: First, the editor queries a given dataset (as a black-box) to find the main concepts, from which the query subject can be selected



RDF Input
From:
http:www.Example1.com
http:www.Example2.com

Query
*Anything*
Author "^Lara"
Year\PubYear > 2007
Title ☑*ArticleTitle*

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

(e.g. `Anything`, `Article`). The editor then finds the possible properties for this subject (e.g. `Title`, `Author`, `Year`). The user selects a property and restricts it using a function (e.g. `MoreThan`) and value (e.g. `2007`); and so on (Section 4). In this way, users can navigate and query a data source without any prior knowledge about it. The symbol "☑" indicates a projection, i.e., appear in the results. When querying multiple sources, two properties (or two instances) are considered the same if and only if they have the same URI. To help end-users not seeing cryptic URI, the editor normalizes URIs by detecting different namespaces of same properties and optionally combines them together (Section 6). In case of different namespaces and property labels (e.g., S1:Year and S2:PubYear), the user can choose the union operator "\" to combine them.

Although MashQL can be used, in a sense, for data integration, but this is not a goal per se. Data integration 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 through datasets) and manually combine them, as shown in the previous example.

**Summary of Contributions:**

- **Query Language** (Section 3). The notational system and constructs that make MashQL an expressive and yet intuitive query language, supporting all constructs of SPARQL.
- **Query Formulation Algorithm** (Section 4). This algorithm is used by the MashQL editor. Its novelty is that it one to navigate through and query a data graph(s) without assuming the end-user to know the schema or the data to adhere to a schema.
- **Graph Signature Index** (Section 5). Because of assumption 2 (data is schema-free), the previous algorithm has to query the whole dataset in real-time, which can be a performance bottleneck because such queries may involve many self-joins. Hence, the interactivity of MashQL might be unacceptable. Thus, we propose a new way of indexing RDF, which we call the *Graph Signature*. The size of a Graph Signature is typically much smaller than the original graph, yielding fast response-time queries.
- **Implementation and Evaluation** (Section 7 and 6). We present two implementations of MashQL: a server-side mashup editor, and a Firefox add-on extension. We evaluate the response-time of MashQL on two large datasets: DBLP and DBPedia; and compare it with Oracle's Semantic Technology. We will show queries can be answered instantly, regardless of the data size.

A preliminary version of MashQL appeared in [23,23], *presenting only the general intuition of MashQL*. This paper is substantially different: a) the intuition is revised; it also includes b) the formal syntax and semantics of MashQL and its mapping into SPARQL, c) the query formulation algorithm, d) the graph-signature index, e) the evaluation and f) the implementation.

## 2. Related Work

Query formulation is the art of allowing people to easily query a data source (e.g., relational database, XML, or RDF). In the background, queries are translated into formal languages (e.g., 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** is the simplest querying method, but it is neither flexible nor expressive. For each query, a form needs to be developed; and changes to a query imply changing its form. Although some methods have been proposed to semi-automate form generation [28] and modification [29] but they generally fail with assumptions 2-4.

**Query-By-Example** A known approach in databases, where users formulate queries as filling tables [50]. However, it requires the data be schematized and the users to be aware of the schema (fails with assumptions 1 and 2).

**Conceptual Queries** As many databases are modeled at the conceptual level using EER, ORM or UML diagrams, one can query these databases starting from their diagrams. Users can select part of a given diagram, and their selection is translated into SQL (ECR [14,41], RIDL[16], LISA[21], ConQuer[11], Mquery[17]). These approaches assume that data has a schema and users have a good knowledge of the conceptual schema (fail with assumptions 1,2,3, and some with 4).

**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 [44], XQuery [33]). 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 – multiple meanings of terms and the mapping between these terms and the elements of a data schema (fails with assumptions 2, 3, and relatively 4).

**Visualize queries.** Several Semantic Web approaches (Isparql[2], RDFAuthor[46], GRQL[9], Nitelight[45]) propose to formulate a SPARQL query by visualizing its triple patterns as ellipses connected with arrows, so that one would need less technical skills to formulate a query. Similarly, some tools had been also proposed to assist formulating XQueries graphically (Altova XMLSpy [1], Stylus Studio [2], Bea XQuery Builder [10], XML-GL [12], QURSED [42]). Although these approaches vary in their intuitiveness they all intend to assist developers - rather than end-users, as they require technical knowledge about the queried sources and their Schemas/DTDs (fail with assumptions 1 and relatively with 2 and 4). In fact, they are close to the query-by-example approaches as they are studio-based query builders, but for semi-structured data.

**Mashup Editors and Visual Scripting.** Some mashup editors (e.g., Yahoo Pipes [7], Popfly [19], sMash [15]) 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 (SparqlMotion[6] and DeriPipes[48]) are inspired by this visual scripting. For example, [48] 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 here, 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 (Section 6). Hence, it is worth noting that the examples of this article cannot be built using Yahoo pipes. Yahoo allows a *limited* support of XML mashups, using scripts in YQL.

**Interactive Queries.** The closest approach to MashQL is Lorel [18], which was developed for querying schema-free XML, and without assuming a user's knowledge about a schema. The difference between them: (First) Lorel partially handles schema-free queries. Like using the Graph-Signature in MashQL, Lorel uses a summary of the data (called DataGuide). However, unlike the Graph Signature, the DataGuide groups unrelated items as they extrinsically use same property labels, which lead to incorrect query formulation. In authors words, "*we have no way of knowing whether O is a publication, book, play, or song. Therefore, a DataGuide may group unrelated objects together*". To resolve this issue, the authors proposed the notion of Strong DataGuide; but the problem is that the size of a *Strong DataGuide* can grow exponentially in case the data is graph-shaped (rather than tree-shaped), thus, can be larger than the original graph: "*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*". (Second) Lorel does not support querying multiple sources (assumption 3); and (Third) its expressivity is basic (assumption 4). MashQL supports path conjunctions, disjunctions, and negation, variables, union, reverse properties, among many others.

Another related approach suggests a highly user **interactive searching** box [37]: a user can write a keyword, the system then smartly and quickly suggests to auto-complete this keyword. We found this approach intuitive as it is simple and does not assume any prior knowledge about the schema indeed (assumption 1). However, unlike MashQL, the existence of a data schema is fundamental to this approach, and this is what makes it highly interactive. The problem also is that this approach cannot play the role of a query language (fails with assumptions 2-4).Being, at the same time, 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 [30] that investigated several query formulation scenario that the casual users prefer. It concluded that a query language should be close to natural language, it should be graphically intuitive, and should not assume prior knowledge about the data. Another recent study [26] has specified similar querying challenges and requirements for in making relational database systems usable for web-based applications.

## 3. THE DEFINITION OF MASHQL

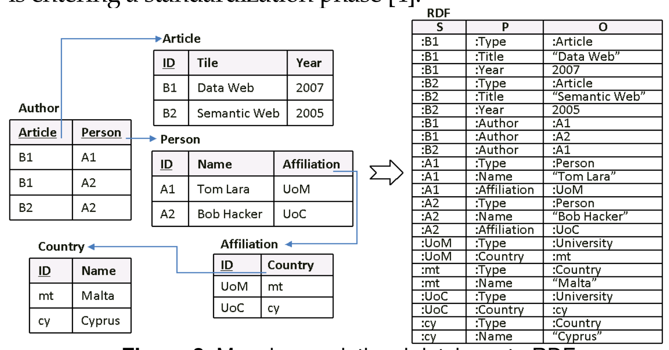
This section defines the data model, the syntax, and the semantics of MashQL. The discussion on how to formulate a query follows in the next section.

### 3.1 The 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. A dataset G is a set of triples <Subject, Predicate, Object>. A subject and a predicate can only be a unique identifier $I$ (URL or a key). An object can be a unique identifier $I$ or a literal $L$.

**Def.1 (Dataset):** *A dataset G is a set of triples, each triple t is formed as <S, P, O>, where* $S \in \mathbf{I}$, $P \in \mathbf{I}$, *and* $O \in I \cup \mathbf{L}$.

The only difference with 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 mapped easily to this *primitive* data model. Figure 3 shows a *simple* example of mapping (or viewing) a database into a 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 relational database and XML into RDF is a mature topic and is entering a standardization phase [4].



Article

| ID | Tile | Year |
|---|---|---|
| B1 | Data Web | 2007 |
| B2 | Semantic Web | 2005 |

Author

| Article | Person |
|---|---|
| B1 | A1 |
| B1 | A2 |
| B2 | A2 |

Person

| ID | Name | Affiliation |
|---|---|---|
| A1 | Tom Lara | UoM |
| A2 | Bob Hacker | UoC |

Country

| ID | Name |
|---|---|
| mt | Malta |
| cy | Cyprus |

Affiliation

| ID | Country |
|---|---|
| UoM | mt |
| UoC | cy |

RDF

| S | P | O |
|---|---|---|
| :B1 | :Type | :Article |
| :B1 | :Title | "Data Web" |
| :B1 | :Year | 2007 |
| :B2 | :Type | :Article |
| :B2 | :Title | "Semantic Web" |
| :B2 | :Year | 2005 |
| :B1 | :Author | :A1 |
| :B1 | :Author | :A2 |
| :B2 | :Author | :A1 |
| :A1 | :Type | :Person |
| :A1 | :Name | "Tom Lara" |
| :A1 | :Affiliation | :UoM |
| :A2 | :Type | :Person |
| :A2 | :Name | "Bob Hacker" |
| :A2 | :Affiliation | :UoC |
| :UoM | :Type | :University |
| :UoM | :Country | :mt |
| :mt | :Type | :Country |
| :mt | :Name | "Malta" |
| :UoC | :Type | :University |
| :UoC | :Country | :cy |
| :cy | :Type | :Country |
| :cy | :Name | "Cyprus" |

**Figure 3.** Mapping a relational database to RDF.

We assume each object literal to have a datatype. If an object value does not have an explicit datatype, it can be implicitly assumed, by taking advantage of XML conventions: the syntax for literals is a String, enclosed in double or single quotes; Integers are written without quotes; Booleans are written as true or false; and so on. Stating a datatype explicitly is done using namespaces, such as: `"1"^^xsd:integer, "2004-12-06"^^xsd:date`.

**Def.2 (Typed Literals):** *A typed literal is a literal object with a tag specifying its Datatype D. Every object literal must have a datatype D: If* $O \in \mathbf{L}$ *then* $O \in \mathbf{D}$.

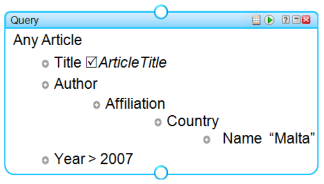
Object literals may also have a language tag $L_t$ (e.g., En, Gr). In the RDF best practice, this is expressed using `@` followed by the tag, such as `"Person"@En, "Ατομο"@Gr`.

**Def.3 (Language Tags):** *A language tag* $L_t$ *is tag optionally associated with a typed literal, to denote to which human language this literal belongs.*

### 3.2 The Intuition of MashQL

A MashQL query $Q$ is seen as a tree. The root tree is called the *query subject* $Q(S)$, which is the subject matter being inquired (see Def.4 in Table 1). A subject can be a particular instance $I$ or a user variable $V$ (see Def.5). Each branch is a *restriction* $R$, on a property of the subject. Branches can be expanded to allow sub trees, called *query paths*. In this case, the value of a property is the subject of sub query. This allows one to navigate through the underlying dataset and build complex queries. As will be explained later, each level a query is expanded it costs a join when this query is executed; thus the deeper the query path is the execution complexity increases.

**Example 2:** To illustrate query paths, we use the data in Figure 3 and seek to retrieve the recent articles from Malta. That is, we retrieve the title of every article that has an author, this author has an affiliation, this affiliation has a country, this country has a name Malta, and the article is published after 2007. This query path can be easily formed and understood in MashQL, as shown in Figure 4.



Any Article
- Title ☑ *ArticleTitle*
- Author
    - Affiliation
        - Country
            - Name "Malta"
- Year > 2007

```
SELECT ?ArticleTitle
FORM < http://localhost.example2#>
WHERE { ?X  :Type  :Article.
        ?X  :Title  ?ArticleTitle.
        ?X  :Author  ?X1.
        ?X1  :Address  ?X2.
        ?X2  :Country  ?X3.
        ?X3 : Name  ?X4.
        ?x  :Year  ?X5.
        FILTER  regex(?X4, "Malta"))
        FILTER  (?X5 > 2007)}
```

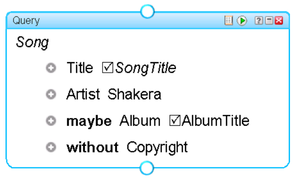
**Figure 4.** Query paths in MashQL and their mappings into SPARQL.

### 3.3 The Syntax and Semantics of MashQL

MashQL queries are not executed directly; instead, they are translated into SPARQL queries, which are submitted for execution. Hence, the semantics of MashQL follow the semantics of SPARQL [43]. 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 (see Def. 6) are retrieved, such that: (i) if a restriction is not prefixed with a modal operator, ($R$:=<*empty*, $P$, $O_f$>), the truth-evaluation of the restriction is considered true if the subject $S$, the predicate $P$, and the object-filter $O_f$ are matched (see the first two restrictions in Figure 5). This case is mapped into a normal graph pattern in SPARQL (see rule-3). (ii) if a restriction is prefixed with the modality "*maybe*" ($R$:=<*maybe*, $P$, $O_f$>), its truth-evaluation is always true (see the 3rd restriction in Figure 5). This case is mapped into an optional graph pattern in SPARQL (see rule 4). (iii) if a restriction is prefixed with the modality "*without*" ($R$:=<*without*, $P$, $Of$>), its 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 5). Notice that there is no such a construct in SPARQL, but in MashQL, we emulate it with an optional pattern and the object $O$ should not be bound (see rule 5).

**Example**. The query in Figure 5 means: retrieve everything (call this thing a *Song*) that: has a title, has 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{
  ?Song :Title ?SongTitle.
  ?Song :Artist :Shakera.
  Optional{?Song :Album ?AlbumTitle}
  Optional{?Song :Copyright ?X1}
  FILTER (!Bound(?X1))}
```

**Figure 5.** A MashQL query and its mapping into SPARQL.

As shown 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 we emulate them (see rules 6-13). MashQL also supports a union 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 retrieves the instances of this subject/object instead of the subject/object itself (See Figure 4). Furthermore, since RDF is a *directed* graph, it is helpful for a user to explore this graph backward. This is supported by the Reverse construct (see Def.10 and rule 20). MashQL also support functions for datatypes, and language tags, sorting, some grouping, which are not presented here for brevity. MashQL support of sorting, distinct, offset, and limit is moved to the *query property* window, which appears by clicking on the top left icon above the query.

To conclude, MashQL is not merely a single-purpose interface, but rather, a general query formulation language, with the four assumptions -introduced earlier- in mind. It is as expressive as SPARQL. Like querying RDF, MashQL can be easily adapted to query XML and relational databases. This can be done by either mapping XML (or RDB) into RDF, or by translating MashQL into XQuery (or SQL).

The design challenge of keeping MashQL an expressive and yet a simple query language is mainly achieved by making *technical variables and namespaces to be implicit*, and especially through *the tree structure of MashQL* queries that hides joins, which is close to the intuition people use in their natural language communication. For example, the query in Figure 4 means, retrieve the article that has an Author $x_1$, $x_1$ has an affiliation $x_2$, and so on. Because the query is represented as a tree, these variables are implicit for end-users. 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 San Francesco , offer a movie called Avatar 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, that are), all of these inquiries can be directly converted into MashQL queries.

**Table 1.** The formal definition of MashQL

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

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

**Def. 6 (Restriction):** *A restriction* $R \coloneqq \langle R_x, P, O_f\rangle$, *$R_x$ is a modal operator,* $R_x \in$ *{empty, maybe, without}; P is a predicate* $(P \in \boldsymbol{I} \cup \boldsymbol{V})$; *$O_f$ is an object filter.*

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

1. $O_f \coloneqq \langle O\rangle$, *where O is an object,* $O \in \boldsymbol{V} \cup \boldsymbol{I}$. This object filter does not add any restriction on the object value as shown in Figure 5.
2. $O_f \coloneqq \langle O, Equals(X, D, L_t)\rangle$, *where X can be a variable or a constant, D is a datatype, and $L_t$ is a language tag.* See rule-6.
3. $O_f \coloneqq \langle O, Contains(X, D, L_t)\rangle$, *O is an object variable, X a regex literal, D a datatype, and $L_t$ a language. O* should be equal to regex(*X*).
4. $O_f \coloneqq \langle O, MoreThan(X, D)\rangle$, *where O is an object variable, X is a variable or a constant, D is a datatype.*
5. $O_f \coloneqq \langle O, LessThan(X, D)\rangle$, *where O is an object variable, X is a variable or a constant, D is a datatype identifier.*
6. $O_f \coloneqq \langle O, Between(X, Y, D)\rangle$, *where X and Y are variables or constants, D is a datatype identifier.*
7. $O_f \coloneqq \langle O, OneOf(V)\rangle$, *where O is an object variable, and V is a set of values* $\{v_1, \ldots, v_n\}$, *$v_i$ is a variable or constant.*
8. $O_f \coloneqq \langle O, Not(f)\rangle$, *where f is one of the functions defined above.* This filter extends all of the above functions with simple negation.
9. $O_f \coloneqq \langle O, Q_i(O)\rangle$, *where O is an object* $(O \in \boldsymbol{V} \cup \boldsymbol{I})$, *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. $O_n = \langle O_1 \backslash O_2 \backslash \ldots \backslash O_n\rangle$, *to indicate unions between objects,* $O_i \in I$.
2. $P_n = \langle P_1 \backslash P_2 \backslash \ldots \backslash P_n\rangle$, *to indicate unions between predicates,* $P_i \in I$.
3. $S_n = \langle S_1 \backslash S_2 \backslash \ldots \backslash S_n\rangle$, *to indicate unions between subjects, where* $S_i \in I$.
4. $Q_n = \langle Q_1 \backslash Q_2 \backslash \ldots \backslash Q_n\rangle$, *to indicate unions between queries,*

**Def.9 (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.10 (Reverse)**: $\langle \sim P\rangle$ *indicates the reverse of the predicate P. Let $R_1$ be a restriction on S s.t.* $\langle S\ P\ O\rangle$, *$R_2$ be* $\langle O \sim P\ S\rangle$, *$R_1$ and $R_2$ have the same meaning.*

**Table 2.** MashQL-To-SPARQL mapping rules

**Rule-1:** The symbol ☑ before a variable means that it will be returned in the results; i.e., included in the SELECT part.

**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, *R* = <*empty, P, $O_f$*>, the mapping: `{S P O}`.

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

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

**Rule 6.** If $O_f = \langle O, Equals(X, D, L_t)\rangle$:
Append the mapping with: `FILTER(?O = X)`
If $D \neq Null$: Append the mapping with: `FILTER(datatype(?O)=D)`
If $L_t \neq Null$: Append the mapping with: `FILTER(lang(?O)= Lt)`

**Rule 7.** If $O_f = Contains(X, D, L_t)\rangle$:
Append the mapping with: `FILTER regex(?O, X)`
If $D \neq Null$: Append the mapping with: `FILTER(datatype(?O)=D)`
If $L_t \neq Null$: Append the mapping with: `FILTER(lang(?O) = Lt)`

**Rule 8.** If $O_f = \langle O, MoreThan(X, D)\rangle$:
Append the mapping with: `FILTER(?O > X)`
If $D \neq Null$: Append the mapping with: `FILTER(datatype(?O=D)`

**Rule 9.** If $O_f = \langle O, LessThan(X, D)\rangle$:
Append the mapping with: `FILTER(?O < X)`
If $D \neq Null$: Append the mapping with: `FILTER(datatype(?O=D)`

**Rule 10.** If $O_f = \langle O, Between(X, Y, D)\rangle$:
Append the mapping with: `FILTER(?O >=X)&& FILTER(?O<=Y)`
If $D \neq Null$: Append the mapping with: `FILTER(datatype(?O)=D)`

**Rule 11.** If $O_f = \langle O, OneOf\ (V)\rangle$: 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 $O_f = \langle O, Not(f)\rangle$: *f* filter is generated as above, but with a negation.

**Rule 13.** If $O_f = \langle O, Q_i(O)\rangle$: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}`.

## 4. QUERY FORMULATION ALGORITHM

We present a novel query formulation algorithm, by which the complexity and the responsibility of understanding a data source (even if it is schema-free) are moved from the user to the query editor. It allows end-users to easily navigate and query an unknown data graph(s). That is, people learn the content and the structure of a dataset while navigating it. The algorithm does not require the data to contain specific information or tags, except being syntactically correct RDF, as discussed in the query model subsection 3.1. Figure 6 shows screenshots of a query formulation scenario.

**Begin**

**<u>Step 0:</u>** *Specify the dataset G* in the Input module. *G* can be one or a merge[2] of multiple data graphs.

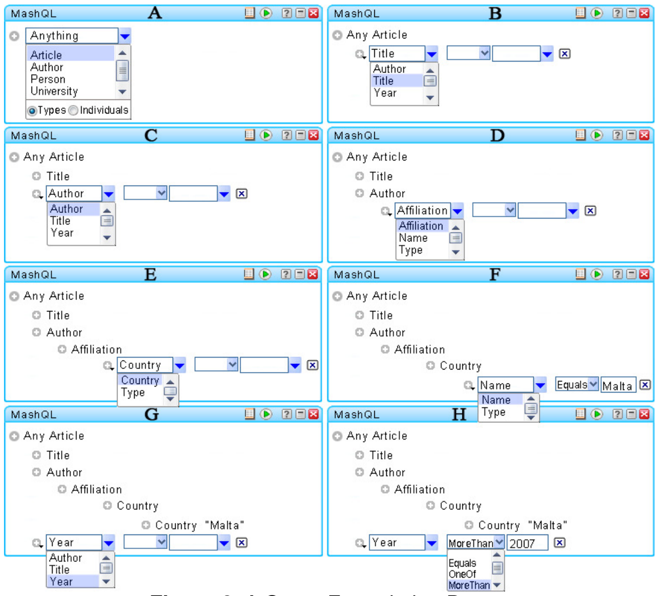


**Figure 6.** A Query Formulation Demo.

**<u>Step 1:</u>** *Select the query subject* $S$, where $S \in S_T \cup S_I \cup V$. That is, after specifying the dataset, users can select $S$ from a drop-down list (Figure 6.A) that contains, either: (**i**) $S_T$: the set of the subject-types in G, such as `Article`; or (**ii**) $S_I$: the union of all subject and object identifiers (i.e., all individuals) in the dataset; or (**iii**) a user-defined *subject label*. In the latter case, the subject is seen as a variable ($S \in V$) and displayed in *italics*; the default subject is the variable label `Anything`. These three options are formalized respectively in relational algebra and SPARQL, as follows:

**(1)** $S \in S_T : \pi_O(\sigma_{P=\text{':Type'}}(G))$
**(1′)** `O1:{(?S1 <:Type> ?O1)}`
**(2)** $S \in S_I : \pi_S(G) \cup \pi_O(\sigma_{O \in URI}(G))$
**(2′)** `S1:{(?S1 ?P1 ?O1)} UNION O1:{(?S1 ?P1 ?O1). Filter isURI(?O1)}`
**(3)** $S \in V$

Users can union the selected subject with another subject(s), e.g., `Author\Person`. After selecting a subject, and then typing the "\" operator, the subject list appears again to select another one(s). The union of all subjects is seen as one subject in the next steps. A union is only possible either between subject-types or individuals, but not a mix of both.

**Repeat Step 2-3** (until the user stops)

**<u>Step 2:</u>** *Select a property P*. Depending on the chosen subject(s) in step 1, a list of the possible properties for this subject is generated (Figure 6.B). There are four possibilities: (**i**) if ($S \in S_T$), such as `Article`, the list will be the set of all properties that the instances of this subject-type have (e.g., `Title`, `Author`, `Year`). (**ii**) if ($S \in S_I$), such as `A1`, the list will be the set of all properties that this particular instance(s) has. (**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. The formalization of these four options are:

**(4)** $(S \in S_T) \rightarrow P \in \pi_{P2}(\sigma_{P1=:Type \wedge O1=Subject}(G) \rtimes_{S1=S2} \sigma(G))$
**(4′)** `P2:{(?S1 <:Type> <S>)(?S2 ?P2 ?O2)}`
**(5)** $(S \in S_I) \rightarrow P \in \pi_P(\sigma_{S=Subject}(G))$
**(5′)** `P1:{(<S> ?P1 ?O1)}`
**(6)** $(S \in V) \rightarrow P \in \pi_P(\sigma(G))$
**(6′)** `P1:{(?S1 ?P1 ?O1)}`
**(7)** $P \in V$

Users can also manually union between properties, in the same way subjects are unioned, such as `Year\PubYear`.

**<u>Step 3:</u>** *Add an object filter on P*. There are three types of filters the user can use to restrict *P*: a filtering function, an object identifier, or a query path. (**i**) A filtering function can be selected from a list (e.g., `Equals`, `MoreThan`, `one of`, `not`); see Figure 6.H. (**ii**) If a user wants to add an object identifier as a filter, a list of the possible objects will be generated. For example, if a user previously chose `Any Article` as a subject, and `Author` as a property, the list of the object identifiers would be {`A1`,`A2`}. The following formalizations specify what the list of object identifiers may contain. Users can also union between objects in the same way subjects and properties are unioned e.g., `A1\A2`.

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

Further, (**iii**) users can also choose to expand the property *P* to declare a path on it (as `Author` in Figure 6.D). In this case, the value `X` of the property `Author`, which is a variable, will be the subject of the sub-query, i.e. a left-join. The possible properties of this subject in the 2nd level will be determined as described in step 2, taking into account all previous selections. The general case of an *n*-level property and *n*-level object (i.e., *n-1* joins) are presented below (14-17) for the cases where the root is a subject-type or a certain instance.

**General Cases**

The n-level paths properties and objects, in case ($S \in S_T$)

**(14)** $P \in \pi_{Pn}(\sigma_{P1=:Type \wedge O1=S}(G) \rtimes_{S1=S2} (\sigma_{C2}(G) \rtimes_{O2=S3} (\sigma_{C3}(G) \ldots \rtimes_{On\text{-}1=Sn}$

2 Merging RDF graphs is straightforward as specified in the W3C standard [42]: all triples are put together; two nodes or two edges are exactly the same iff they have the same labels (i.e., URI).

$(\sigma_{Cn}(G)))))$

**(14′)** `Pn:{(?S1 <Type> O)(?S1 P2 O2)(O2 P3 O3) … (On-1 ?Pn ?On)}`

**(15)** $O \in \pi_{On}(\sigma_{P1=:Type \wedge O1=S}(G) \ltimes_{S1=S2} (\sigma_{C2}(G) \ltimes_{O2=S3} (\sigma_{C3}(G) \ldots \ltimes_{On\text{-}1=Sn} (\sigma_{Cn}(G)))))$

**(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)** $P \in \pi_{Pn}(\sigma_{C1}(G) \ltimes_{O1=S2} (\sigma_{C2}(G) \ltimes_{O2=S3} (\sigma_{C3}(G) \ldots \ltimes_{On\text{-}1=Sn} (\sigma_{Cn}(G)))))$

**(16′)** `Pn:{(S1 P1 O1)(O1 P2 O2 … (On-1 ?Pn ?On)} \\Subject ∈ SI`

**(17)** $O \in \pi_{On}(\sigma_{C1}(G) \ltimes_{O1=S2} (\sigma_{C2}(G) \ltimes_{O2=S3} (\sigma_{C3}(G) \ldots \ltimes_{On\text{-}1=Sn} (\sigma_{Cn}(G)))))$

**(17′)** `On:{(S1 P1 O1)(O1 P2 O2)(O2 P3 O3)…(On-1 Pn ?On)}`

**<u>Step 4:</u>** The symbol ☑ before a variable is used to indicate that it will be returned in the results (i.e., projection).

**End.**

This algorithm illustrates how users interact with the MashQL editor and formalizes the "background queries" that need to be executed in each interaction. In this way, users can navigate and query a data graph without prior knowledge about it, even if it is schema-free. Section 6 implements this algorithm in two different editors, and discusses implementation issues to further enhance the query formulation process in case of large and cryptic data. Next, we focus on the performance of this algorithm.

## 5. Graph Indexing: The Graph-Signature

One of our key assumptions for querying the Data Web is that data is schema-free. This is indeed a challenging requirement for query formulation as the editor's background queries need to be executed on the *whole* dataset and in *real-time*, because there is no offline schema that can be used instead. In such a user-interaction setting, the response-time is an important factor that needs to be taken into consideration and which should be small, preferably within 100 ms [35]. Achieving such a short interaction time for background queries with *graph-shaped data* is even more challenging, because the exploration of a graph stored in a *relational* table G(S,P,O) can be expensive as this table needs to be self-joined many times [8]. A query with *n* levels involves *n-1* joins. Pre-computing and materializing all possible MashQL's background queries is not an option since the space requirements are too high; thus an efficient RDF indexing is needed. Several approaches have been proposed to index RDF, such as Oracle[3] [13], C-Store[4] [8], and RDF3X[5] [39]. Although these approaches have shown good performance –a query with a medium complexity costs some seconds- however, this performance is unacceptable for an interactive query formulation session, especially in the case of large graphs.

In this section, we present the *Graph Signature*, a novel approach for indexing RDF graphs, which is a complementary rather than an alternative to the approaches mentioned above. Our goal is *not* to optimize any arbitrary RDF query, but rather to enhance the performance of the background queries presented in the previous section. Next –before presenting the Graph Signature-, we generalize the background queries into one *query model*. The rest of the section shows how this query model is significantly optimized using the Graph Signature.

### 5.1 The Query Model

As one may notice, each of the 17 background queries formalized earlier is a query path, i.e., a linear-shaped query. Star-shaped and tree-shaped queries are not needed in query formulation. We define a query path as an expression of the form: $\{O_1\ P_1\ O_2\ P_2\ \ldots\ P_n\ O_n\}$, where $O_i$ is a node, and $P_i$ is an edge. Both nodes and edges can be variables. A variable node is denoted as $?O_i$ and a variable edge as $?P_i$. A query can return either a node or an edge. For query formulation, we only need to retrieve the last node/edge in the path; that is, we need to retrieve either the edge $P_n$ or the node $O_n$. Hence, the *query model* is formed as: $O_n | P_n$:$\{O_1\ P_1\ O_2\ P_2\ \ldots\ P_n\ O_n\}$. For example, the query P:{B2 Author ?O1 ?P ?O2} retrieves the properties of the authors of B2; and O:{B2 Author ?O1 Affiliation ?O} retrieves the affiliations of the authors of B2. Each of the 17 background queries in the previous section is a special case of this query model; hence, optimizing the query model is an optimization of all background queries.

### 5.2 The Intuition of the Graph Signature

The idea of the Graph Signature is to summarize a given RDF graph, so that the background queries can be answered from this summary. Because the size of the summary is smaller than the original graph, queries can be faster. Given an RDF graph G, its Graph Signature $S$ is a twofold summary: the O-Signature $S_O$ and the I-Signature $S_I$. $S_O$ is a summary of the original graph such that nodes that *have the same outgoing paths* are grouped together. $S_I$ summarizes a graph by grouping nodes that *have the same incoming paths*, which is analogous to the 1-index [36].

**Example.** Figure 7 provides an example of an RDF graph and its O/I-signatures. In this example, {A2, A3} are grouped in the $S_O$ because they have the same outgoing paths until the end. A1 is not part of this grouping as it does not have the path Affiliation.Student. In the I-Signature, A4 is not grouped with {A1, A2, A3} as it has different incoming paths, e.g., Student. Each of the two summaries is computed and stored separately, but they are jointly used to produce precise answers, as will be discussed shortly .

Let us now query these signatures, and compare their results with the results obtained from the original graph *G*. We call the answer of *G*, the *target answer*. Figure 8 shows examples of queries and their answers.

[3] 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.

[4] C-Store [8] suggested partitioning the RDF table vertically, into *n* two-column tables, where *n* is the number of unique properties in the data.

[5] RDF3X [39] to only build many B+-Tree indexes, and a "*careful optimization of complex join queries*".

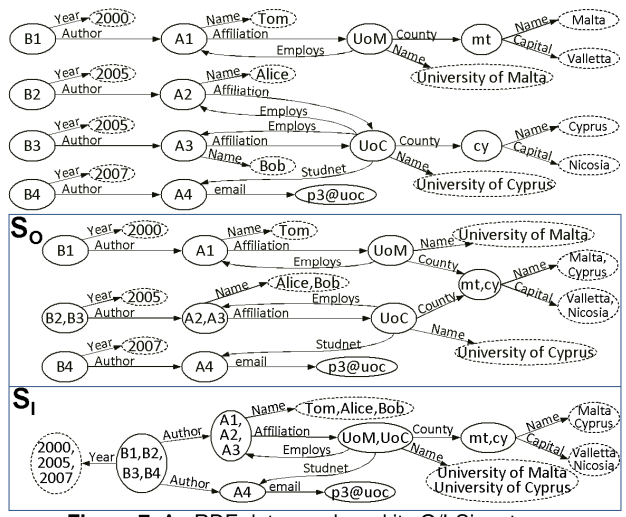


**Figure 7.** An RDF data graph and its O/I-Signatures.

| | Query | $S_O$ Answer | $S_I$ Answer | $S_O \cap S_I$ | G Answer |
|---|---|---|---|---|---|
| Q1 | P:{B2 Author ?O1 ?P ?O2} | Affiliation, Name | Affiliation, Name, email | Affiliation, Name | Affiliation, Name |
| Q2 | P:{B2 Author ?O1 Affiliation ?O2 ?P ?O3} | Name, Country, Employs, Student | Name, Country, Employs, Student | Name, Country, Employs, Student | Name, Country, Employs, Student |
| Q3 | P:{UoM Student ?O1 ?P ?O2} | { } | email | { } | { } |
| Q4 | O:{B2 Author ?O1 Affiliation ?O} | UoC | UoM, UoC | UoC | UoC |
| Q5 | O:{?O1 Author ?O2 Affiliation ?O} | UoM, UoC | UoM, UoC | UoM, UoC | UoM, UoC |

**Figure 8.** GS answers compared with the target answers.

As shown in Figure 8, each part of the Graph Signature produces the correct answer and some more results, called *false positives*. That is, the target answer is equal to or a subset of the answer of each part. Hence, the intersection of the $S_O$ and $S_I$ answers equals or is a small superset of the target answer. We shall show in subsection 5.8 how false positives (if any) are eliminated, in order to *always* achieve *precise answers.* Hence, instead of evaluating the background queries on the original graph, we evaluate them on the Graph Signature. Because the size of the graph signature is much smaller than the original graph, querying it is much faster. Subsection 5.9 positions the novelties of the Graph Signature w.r.t. related work. In Section 7, we present an evaluation of MashQL's background queries over the Graph Signature of large datasets (DBLP and DBPedia), and show that it yields to an instant user-interaction, regardless of the complexity of the background queries.

In the next subsections, we turn our focus to formally define the Graph Signature and its construction and storage. We shall come back again (subsection 5.8) to discuss how the background queries are evaluated on the Graph Signature.

## 5.3 The Notion of Bisimilarity

Since each node in the Graph Signature is in fact an *equivalent class* of some nodes in *G*, one way to compute the Graph Signature is a full traversal of *G*. For example, we take every node in *G*, compute all outgoing paths from this node, and compute all incoming paths into this node. Then, we construct the O-Signature by grouping the nodes having the same outgoing paths; and similarly the I-Signature. This way is called *trace equivalence* [20]. Unfortunately, this way is computationally expensive and known to be PSPACE-complete [46]. The solution (as suggested by [32]) is to use the notion of *bisimilarity,* which is extensively discussed in the literature of process algebra [20,40] and which *implies trace equivalence*. The idea of bisimilarity, in RDF terms, is to group nodes having the same properties, and then iterate; at each iteration step we split a group of nodes if it violates bisimilarity. We repeat until our groupings are stable. Next, we adopt the typical definition of bisimilarity [40] and modify it to suite RDF graphs.

**Definition (*O*-Bisimilarity $\approx_O$)**

*O*-Bisimilarity is a symmetric binary relation $\approx_O$ on *G.* Two nodes $S_1$ and $S_2$ are *O*-bisimilar ($S_1 \approx_O S_2$), if and only if:

i. The set of the property labels of $S_1$ equals the set of the property labels of $S_2$. In RDF terms, there exists ($S_1$ $P_1$ O)… ($S_1$ $P_m$ O), and ($S_2$ $P_1$ O) … ($S_2$ $P_n$ O), such that, the distinct set of properties of $S_1$ {$P_1$..,$P_m$} equals the distinct set of properties of $S_2$ {$P_1$,..,$P_n$}.

ii. If $S_1'$ is a successor of $S_1$ through a property $P_i$ ($S_1 \xrightarrow{P_i} S_1'$), and $S_2'$ is a successor of $S_2$ through a property $P_i$ ($S_2 \xrightarrow{P_i} S_2'$) then $S_1' \approx_O S_2'$, and $S_2' \approx_O S_1'$.

**Definition (*I*-Bisimilarity $\approx_I$)**

*I*-Bisimilarity is a symmetric binary relation $\approx_I$ on *G.* Two nodes $S_1$ and $S_2$ are *I*-bisimilar ($S_1 \approx_I S_2$), if and only if:

i. The set of the property labels *into* $S_1$ equals the set of the property labels into $S_2$. That is, there exist (O $P_1$ $S_1$)… (O $P_m$ $S_1$), and (O $P_1$ $S_2$) … (O $P_n$ $S_2$), such that, the set of properties into $S_1$ {$P_1$..,$P_m$} equals the set of properties into $S_2$ {$P_1$,..,$P_n$}.

ii. If $S_1'$ is a predecessor of $S_1$ through a property $P_i$ ($S_1' \xrightarrow{P_i} S_1$) and $S_2'$ is a predecessor of $S_2$ through a property $P_i$ ($S_2' \xrightarrow{P_i} S_2$), then $S_1' \approx_I S_2'$, and $S_2' \approx_I S_1'$.

## 5.4 The Definition of the Graph Signature

**Definition (Graph Signature).** Given an RDF graph *G*, its Graph Signature *S* is comprised of two summaries: O-Signature $S_O$ and I-Signature $S_I$. In short, $S = <S_O, S_I>$.

**Definition (O-Signature).** Given an RDF graph *G*, its $S_O$ is a directed labeled graph, such that, each node in $S_O$ is an equivalent class ($\approx_O$) of some nodes in *G*; and each edge *p* in $S_O$ from *u* to *v* ($u \xrightarrow{P} v$) iff *G* contains an edge *p* from *a* to *b* ($a \xrightarrow{P} b$) and $a \in u, b \in v$.

**Definition (I-Signature).** Given an RDF graph *G*, its $S_I$ is a directed labeled graph, such that, each node in $S_I$ is an equivalent class ($\approx_I$) of some nodes in *G*; and each edge *p* in $S_I$ from *u* to *v* ($u \xrightarrow{P} v$) iff *G* contains an edge *p* from *a* to *b* ($a \xrightarrow{P} b$) and $a \in u, b \in v$.

## 5.5 Construction of the Graph Signature

To compute the Graph Signature, we use the standard algorithm for computing bisimilarity [40]. We modify this algorithm to suit RDF graphs, for computing both the O-Signature and the I-Signature (see Figure 9). The input in each algorithm is a data graph and the output is the O/I-Signature. As mentioned earlier, to compute the O-Signature, first we group nodes having the same immediate properties; then we iterate -to split groupings that are not O-bisimilar- until all groupings are stable. As shown in steps 5-7, an equivalent class *A* is *stable* iff for every path *P* from *A* into another group-node *B*, each instance of *A* has a successor in *B*. In other words, let *X* be the predecessors of *B* through *P*, in *G*; then *A* should be a subset of or equal to *X*. Otherwise, *A* should be split into two nodes: ($A \cap X$) and ($A$ -

*X*). The same (but the opposite) way is used to compute the I-Signature. As discussed in [40] the maximal time needed to compute bisimilarity for this algorithm is $O(m \log n)$, where $m$ is the number of nodes and $n$ is the number of edges. Hence, the time complexity of computing the overall Graph Signature is $O(m \log n)$.

```
Procedure ComputeOSignature(G,S_O)
begin
1.  S_O = (a copy of) G.      Stable = false
2.  Group nodes having the same property labels. \\the initial step
3.  while (Stable ≠ True) do   // iterate until the grouping is stable
4.    foreach node A in S_O do
5.      foreach path P_i from A into a node B do
6.        X = P_i^-1(ext[B]^G)  //find the predecessors of B through P_i, in G
7.        if (A ⊈ X) then replace A  by (A ∩ X) and (A - X)   //split A
8.    if there was no split then Stable=True
end
Procedure ComputeISignature(G,S_I)
begin
1.  S_I = (a copy of) G.      Stable = false
2.  Group nodes having the same incoming properties \\the initial step
3.  while (Stable ≠ True) do //iterate until the grouping is stable
4.    foreach node A in S_I do
5.      foreach P_i path into A from a node B do
6.        X = P_i(ext[B]^G)       //find the successor of B through P_i, in G
7.        if (A ⊈ X) then replace A  by (A ∩ X) and (A - X)   //split A
8.    if there was no split then Stable=True
end
```

**Figure 9.** An algorithm to compute the Graph-Signature[40].

## 5.6 Storage of the Graph Signature

Since each part of the Graph Signature is a directed labeled graph it is convenient to store them in the same way the data graph is stored. In our implementation, a data graph and its O/I-signatures are stored and queried in the same way, using the Oracle Semantic Technology (see Section 6). To store the extent of the O/I-signature, each node in $S_O$ and $S_I$ is given an identifier, and this *id* is used in a lookup table to store the nodes in *G* belonging to their equivalent classes in $S_O$ and $S_I$, ext(SoID, SiID, Node). This table is called the *extent* of the graph signature. A full-text index can be built on this table for keyword search, and statistics can be maintained for query optimization purposes. For query formulation, we only store node labels and their group *id*s in a table as specified above.

## 5.7 The Size of the Graph Signature

The space cost to store each part of the Graph Signature consists of the space of the signature and the space of its extent. The size of each part of the Graph Signature is at most as large as the data graph; but in practice, it is much less, as our evaluations show. The size of the extent is exactly the number of unique nodes in the data graphs. In the following we present some techniques that yield a significant reduction of the overall size of the Graph Signature:

1. *Literal nodes* can be excluded, as they are not used in query formulation. We assign literal nodes to *null* before computing the Graph Signature.
2. *Annotation properties* can be excluded. There are several types of properties in RDF that are not intended to represent data, but rather, to describe data, such as rdf:Description, rdf:comment, rdf:label, rdf:about, or rdfs:seeAlso.
3. *Synonym properties* can be joined. Because of different namespaces and conventions, it is likely that different properties have the same semantics (e.g., foaf:FirstName and foaf:GivenName, foaf:mbox and :email). Such properties can be joined by replacing them with a chosen label.
4. *Equivalence properties* can be normalized. Certain properties indicate that the subject and object in a triple are equal, such as rdf:SameAs and rdf:Redirect. Normalizing these properties can be done by assigning the subject and the object the same URI.
5. *Certain properties* can be excluded. We may wish to exclude some properties that are not likely to be queried, such as LongAbstract in DBPedia.

Before computing the graph signature, we process a *configuration file*, which we have built for the properties to be excluded, joined, or normalized.

**A special case** property is the rdf:Type. As this property is likely to be used in query formulation, it should be well-indexed. For this, we extend the lookup table, which we use to store the extents. Instead of having the lookup table as ext(SoID, SiID, Node), we have ext(Type, SoID, SiID, Node). Hence, we can look up not only the group of a node, but also the node(s) and the group(s) of a certain type.

## 5.8 Evaluating Queries with the Graph Signature

As discussed earlier, the answer obtained from the O-Signature -and similarly the I-Signature- is always a superset or equals the target answer (the answer obtained from the data graph). In case the answer of the O/I-signature equals the target answer, we call it a *precise answer*; otherwise, it is called a *safe answer*, since it equals the target answer and some *false positives*. The intersection of the answers of both the O-Signature and the I-Signature is a smaller superset or equals the target answer. The following theorems state when the Graph Signature produces precise and safe answers; the proofs are sketched in the appendix.

**Theorem 1.** *Given a query, the answer of the O-Signature is always safe; and similarly the answer of the I-Signature.*

**Theorem 2.** *Given any query retrieving edge labels, the answer of the O-Signature is always precise.*

**Theorem 3.** *Given a query, with all nodes variables, the answer of the I-Signature is always precise.*

**Theorem 4.** *Given a query, if the answer of the O/I-Signature is empty or the intersection of both is empty, then this answer is always precise.*

Based on these theorems, the flowchart in Figure 10 depicts the evaluation scenario. Given a background query Q, if $P_n$ is projected (i.e., the last edge label is retrieved), it can be precisely answered from the O-Signature, as stated in theorem 2. Examples of such queries follow: P:{?O ?P ?O1}; P:{?O Author ?O1 ?P ?O2}; P:{?O Author ?O1 Affiliation ?O2 ?P ?O3}. This case represents the majority of the background queries in query formulation, as it allow one to navigate through and understand the structure of a data graph.

In case a background query projects $O_n$ (i.e., the last node label is retrieved), and all node labels ($O_1$ …$O_n$) in the query are variables, such as O:{?O1 Author ?O2 Affiliation ?O}, the answer of the I-Signature is precise (see theorem 3). However, if some nodes in the query are not variables, such as B2 in the query O:{B2 Author ?O1 Affiliation ?O}, the answer of the O-Signature -and the answer of the I-Signature- is safe. In fact, the more variable nodes a query contains the less false positives are produced. To reduce the number of false positives in this case, we evaluate the query on the O-Signature and the I-Signature separately, and we intersect both results. If the intersection is empty (or one of the answers in empty), then the answer is precise (see theorem 4). Otherwise, the intersection of both answers is a small superset or equals of the target answer. Such results might be sufficient indeed in the query formulation practice; otherwise, to eliminate the false answers, we evaluate the query on the data graph, and optimize it using the intersection of both answers. The idea of this optimization is to simply join the results of the intersection with the data graph, and execute the query on this join. That is, the false results are eliminated as they do not satisfy the query on the data graph.

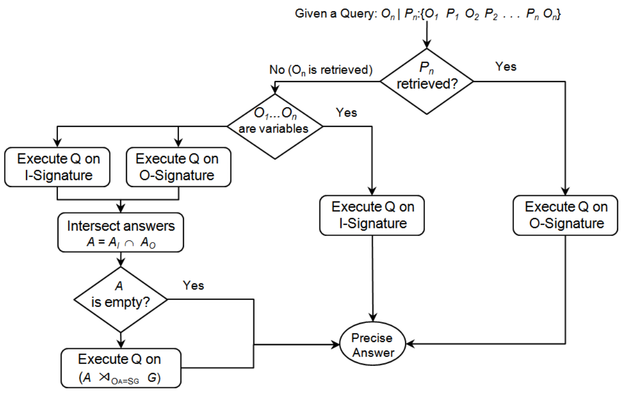

Given a Query: $O_n$ | $P_n$:{$O_1$ $P_1$ $O_2$ $P_2$ . . . $P_n$ $O_n$}
$P_n$ retrieved?
No ($O_n$ is retrieved)
Yes
$O_1$...$O_n$ are variables
Yes
Execute Q on I-Signature
Execute Q on O-Signature
Intersect answers $A = A_I \cap A_O$
Execute Q on I-Signature
Execute Q on O-Signature
$A$ is empty?
Yes
Precise Answer
Execute Q on ($A \rtimes_{OA=SG} G$)

**Figure 10.** Depiction of the Execution Plan.

We have implemented the query evaluation scenario described above (i.e., execution plan) in a table function in Oracle 11g. This function takes a query as input and produces precise results as output. The function first parses the query to find the constant node labels -that are not variables- and replace them with their group IDs. For example, the query P:{B2 Author ?O1 ?P ?O2} is re-written as P:{123 Author ?O1 ?P ?O2}, where 123 is the group id of the B2. The function then checks whether the query is retrieving edges or has all nodes as variables, if so, the function then executes it on the O-Signature or I-Signature respectively. Otherwise, it executes it on the O-Signature and I-Signature in parallel, and intersects both results. If the result is not empty, the function eliminates the possible false positives by executing the query on the join of the data graph and the intersection, as descried earlier.

Because the time-complexity of evaluating a query is preoperational to the size of the graph [36], evaluating queries on the Graph Index yields a better performance, since its size is likely to be much smaller than the data graph. See our evaluation is Section 7.

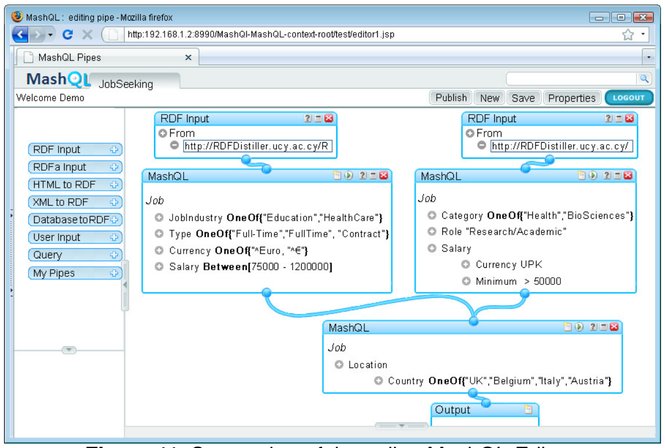
### 5.9 Related Work to the Graph Signature

The notion of *structural summaries* has been proposed to summarize XML data, for XQuery optimization. The **DataGuide** [38] was the first to suggest summarizing XML by grouping nodes reachable by *any* incoming path. The problem with this way is that, because nodes that extrinsically have some similar property labels are grouped together, many false positives are generated. The **Strong DataGuide** [18] proposed to solve this issue by grouping nodes reachable by simple paths, as the DataGuide; but, it allows a node to exist in multiple groups. As pointed by the authors, this approach is efficient for tree-shaped data, but the size of the summary grows exponentially the more the data is graph-shaped (and can be larger than the original graph). The **1-index** [36] proposed to group nodes reachable by *all* incoming paths (which is analogous to our I-Signature), but it does not consider the outgoing paths (as our O-Signature) that yields an efficient reduction of false positives. A similar approach to the 1-index (called **A(*k*) index** [32]) suggested to also group nodes reachable by all incoming paths (but up to *k* levels), thus it can only answer queries with *k* levels. Since this approach generates many false positives, the same authors of the A(*k*) suggested later another approach called **F&B index** [31]. This approach groups nodes reachable by both all incoming *and* all outgoing paths, i.e., forward and backward at the same time. This approach produces much less false positives indeed, but its size is *not* much less than the original. For example, the size of the F&B index for the Xmark dataset is only 10% less than the original [31]. As such, the time needed to query the F&B summary is close to querying the original graph. Furthermore, all of the above approaches cannot be applied for RDF because (i) RDF is graph-shaped rather than tree-shaped; hence applying them produces large-size indexes; and (ii) XML queries are not the same as RDF queries (i.e., different query models). For example, in XML we typically retrieve node labels, but in RDF, we also need to retrieve property labels.

The novelty of our graph index is: (i) the bisimilarity algorithm is adapted to suite RDF graphs, s.t. it is not necessary for a node to have unique outgoing edges, as in XML; (ii) unlike the F&B approach that generates *one* large incoming-and-outgoing index in order to generate less false positives, we store the incoming and outgoing indexes separately, but they are jointly used, thus achieving small indexes and less false positives at the same time; and (iii) a query model and an evaluation scenario for RDF query paths is proposed, which is different from XML paths, as property labels, not only node labels, can be retrieved.

## 6. IMPLEMENTATION

We implemented MashQL in two scenarios: an online server-side query and mashup editor, and a browser-side Firefox add-on editor. The former is illustrated in Figure 11 and Figure 12. Its functionality 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 and stored in XML; v) a module that retrieves, merges, and presents the results of the submitted SPARQL query. MashQL queries can be materialized and published if needed. Each published query is given a URL, and its output is seen as a concrete RDF source.



RDF Input
From
http://RDFDistiller.ucy.ac.cy/R

RDF Input
From
http://RDFDistiller.ucy.ac.cy/

MashQL
*Job*
JobIndustry **OneOf**{"Education","HealthCare"}
Type **OneOf**{"Full-Time","FullTime", "Contract"}
Currency **OneOf**{"^Euro, "^€"}
Salary **Between**[75000 - 1200000]

MashQL
*Job*
Category **OneOf**{"Health","BioSciences"}
Role "Research/Academic"
Salary
Currency UPK
Minimum > 50000

MashQL
*Job*
Location
Country **OneOf**{"UK","Belgium","Italy","Austria"}

**Figure 11.** Screenshot of the online MashQL-Editor.

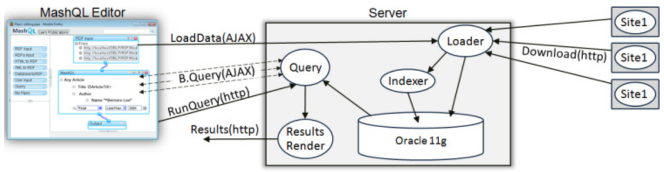


**Figure 12.** System Model.

When a user specifies an RDF data source(s) as input, it is bulk-loaded into an Oracle 11g, and its Graph Signature is constructed. Subsequently, the MashQL Editor uses AJAX to dispatch background queries and the SPARQL translation of formulated MashQL queries for execution by the Oracle 11g. We chose Oracle 11g because of its support for native RDF queries and storage.

As one may notice, MashQL's GUI follows the style Yahoo Pipes visualizes feed mashups, and uses the Yahoo Pipes's 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 piping web feeds***. It is worth noting also that the examples of this article cannot be built using Yahoo pipes, as it does support querying structured data. 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. In fact, Yahoo Pipes, as well as Popfly and sMash, are motivating for -rather than solving- the problem of structured-data retrieval.

In an alternative implementation, we developed the MashQL editor as an add-on to the Firefox browser. This extension has the same functionalities of the online editor. However, no databases or RDF indexing are used for storing, indexing, and querying the data sources, but rather, the SPARQL queries are executed inside the browser, using the JENA SPARQL query libraries. Hence, the size of the input sources is limited to the client's memory. The goal of this Firefox extension 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***.

We refer the reader to technical report [25] for more technical details and MashQL use cases.

### 6.1 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 6). 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.

## 7. Evaluation

This section presents three types of evaluations: (i) the scalability of the Graph Signature, (ii) the time-cost of formulating a MashQL query using the Graph Signature, and compare it with using the Oracle Semantic Technology; and (iii) the usability of the MashQL editors.

### 7.1 Datasets and Experimental Settings

Our evaluation is based on two public datasets: A) DBLP and B) DBPedia. The DBLP (700MB size) is a graph of 8 million edges. We partitioned this graph into three parts: A8 is the whole DBLP; A4 is 4 million triples from A8; and A2 is 2 millions. No sorting is used before the partitioning. Figure 13 shows some statistics. The DBPedia (6.7 GB) is a graph of 32 million edges, which is an RDF version of the Wikipedia. Similarly, DBPedia is partitioned into 3 parts. We choose these datasets in order to illustrate the scalability of our Graph Index in case of homogenously and heterogeneously structured graphs. DBLP is more homogenous, as most of its nodes have similar paths.

However DBPedia is known to be a noisy collection of RDF triples. Each of the 6 partitions is loaded into a separate RDF model in Oracle 11g, which was installed on an server with 2GHz dual CPU, 2 GB RAM, 500GB HHD, 32-bit Unix OS.

| Number of | (A)DBLP | | | (B)DBPedia | | |
|---|---|---|---|---|---|---|
| | A8 | A4 | A2 | B32 | B16 | B8 |
| Unique Triples | 9M | 4M | 2M | 32M | 16M | 8M |
| Unique Subjects | 1.1M | 1M | 0.8M | 9.4M | 6M | 4M |
| Unique Predicates | 28 | 27 | 26 | 35 | 35 | 34 |
| Unique Objects | 2.4 | 1.2 | 0.7M | 16M | 8.7M | 4.7M |
| Data Size | 700MB | 350MB | 170MB | 6.7GB | 3.1GB | 1.4GB |

**Figure 13.** Statistics about the experimental data.

## 7.2 Scalability Evaluation

We built an O-signature and I-Signature for each partition (see Figure 14). As one can see, the time cost to build the $S_O$ and $S_I$ is linear with respect to the data size. For example, for $S_O$, B2 (2M triples) costs 48 seconds, the time is almost doubled when the data size is doubled.

| | Number of | (A)DBLP | | | (B)DBPedia | | |
|---|---|---|---|---|---|---|---|
| | | A8 | A4 | A2 | B32 | B16 | B8 |
| $S_O$ | Indexing Time (Sec) | 219 | 90 | 48 | 563 | 194 | 106 |
| | Equivalence Classes | 4K | 28K | 12K | 103K | 110K | 56K |
| | Triples in O-Signature | 34K | 190K | 62K | 1M | 686K | 244K |
| $S_I$ | Indexing Time (Sec) | 83 | 40 | 18 | 641 | 293 | 142 |
| | Unique Categories | 61 | 14 | 43 | 14K | 7K | 3K |
| | Triples in O-Signature | 108 | 72 | 62 | 84K | 30K | 10K |

**Figure 14.** The O-Signature for all partitions.

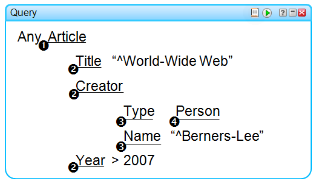
What is more scalable, is the behavior of the index with respect to the number of the triples. For example, the whole DBLP A8 (8M triples) is summarized in $S_O$ by only 34K triples; this number is larger when the data is smaller, 190K for A4. This is because (although we did not apply any sorting before partitioning the data, but) more similarities were found when the whole data is put together. In other words, some nodes in A4 are grouped in several equivalence classes (instead of one) as they have different paths, while when all data is put together in A4, it is found that these nodes have the same paths. This implies that the size of the Graph Signature does not necessarily increase if more triples are added to the data graph. The size of the O-Signature reflects the homogeneity of a graph. For example, the O-Signature for A8 (34K) is smaller than the O-Signature for B8 (244K), as DBLP is more homogenous. Nevertheless, for both datasets, the generated O/I-signatures fit in a small memory, thus joining it many times still yields fast querying as we show next.

The I-Signature happens here to be smaller than the O-Signature. The reason is that root nodes (which are many, in DBLP and DBPedia) are all grouped together in one equivalent class.

## 7.3 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 this article; 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 present three MashQL queries. We identify the set of background queries, and evaluate them on both: (1) Oracle's Semantic Technology, which is the native RDF index[6] in Oracle 11g [13]; and (2) the Graph Signature index (as described in subsection 5.8). We also store the Graph Signature in Oracle 11g as described subsection 5.6.

**Experiment 1:** To formulate the query in Figure 15 on DBLP, the user first selects the query subject from a list. The query that produces this list is annotated by ❶. The user then selects a property of this subject from a list. The query that 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 Semantic Technology. The cost[7] (in seconds) is shown in Figure 16.



```
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)
```

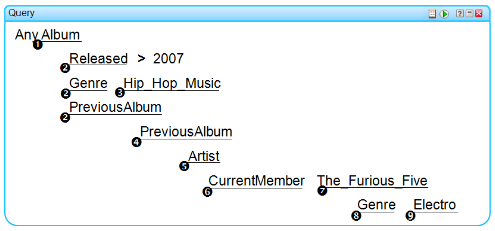
**Figure 15. 4** Queries are needed to formulate this MashQL query.

| Query | (A8) 8 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. 871 | 0.001 | 0.546 | 0.001 | 0.471 |
| $Q_4$ | 0.001 | 1.208 | 0.001 | 0.835 | 0.001 | 0.650 |

**Figure 16.** Time cost (in seconds) of background queries.

As shown by this experiment, the time cost for each query remains within few milliseconds using the Graph Signature, regardless of the data size and complexity of the query. This is because the size of the Graph Signature is small, if 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 17).



```
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 ?O)
❽ P:(?S Type Album)(?S PreviousAlbum ?O1) (?O1 PreviousAlbum? ?O)
     (?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 ?O)
```

[6] As described earlier, Oracle [13] stores RDF triples in one table G(s,p,o); thus a query with *n*-levels implies joining the table *n*-1 times. To improve the querying performance, Oracle proposed to build several B-tree indexes on G, as well as to build subject-property materialized views on G, such as $V_1$(s, $p_1$, $p_2$, …$p_n$). A tuple in $V_1$ is a subject identifier x, and the value of the column $p_i$ is an object y. In this way, data is transformed -somehow- from a graph form into a relational form; thus, less number of joins when executing a query. These subject-property views are seen as auxiliary, rather than core, indexes. This is because there is no general criteria to know which subjects and which properties to group. Oracle uses statistics to find possibly good groupings (i.e., views), otherwise, queries are executed one the original G data graph; hence queries with many joins remain a challenge.

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

**Figure 17.** 9 queries are needed to formulate this MashQL query.

| Query | (B32) 32 M | | (B16) 16 M | | (B8) 8 M | |
|---|---|---|---|---|---|---|
| | GS | Oracle | GS | Oracle | GS | Oracle |
| $Q_1$ | 0.003 | 0.017 | 0.003 | 0.012 | 0.003 | 0.008 |
| $Q_2$ | 0.002 | 172 | 0.002 | 118 | 0.002 | 85 |
| $Q_3$ | 0.005 | 859 | 0.004 | 592 | 0.003 | 423 |
| $Q_4$ | 0.005 | 2576 | 0.004 | 1776 | 0.004 | 1269 |
| $Q_5$ | 0.005 | 3864 | 0.004 | 2665 | 0.004 | 1903 |
| $Q_6$ | 0.005 | - | 0.005 | - | 0.005 | - |
| $Q_7$ | 0.007 | - | 0.007 | - | 0.007 | - |
| $Q_8$ | 0.005 | - | 0.005 | - | 0.005 | - |
| $Q_9$ | 0.007 | - | 0.007 | - | 0.007 | - |

**Figure 18.** Time cost of the background queries in Figure 17. Queries taking more than 5000sec were terminated and their time is not reported.

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 expose the limits of both our Graph Signature and Oracle's index. Figure 19 shows a query where all nodes and properties are *variables*. It means, what are the properties of the properties of … (at 9 level) of properties of anything. After selecting the query subject as the variable *Anything*, 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: `P:(?Anything ?RelatedTo1 ?O1)(?O1 ?P ?O2);` at the 3rd level ❸: `P:(?Anything ?RelatedTo1 ?O1)(?O1 ?RelatedTo2 ?O2)(?O2 ?P ?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*.

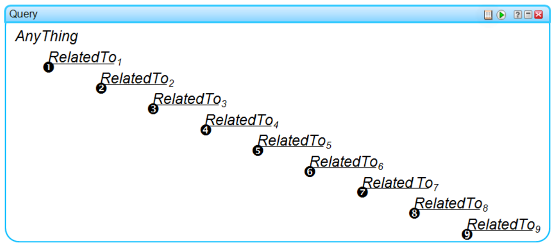


**Figure 19.** A query with all predicates are variables.

| Query | (B32) 32 M | | (B16) 16 M | | (B8) 8 M | |
|---|---|---|---|---|---|---|
| | GS | Oracle | GS | Oracle | GS | Oracle |
| $Q_1$ | 0.003 | 0.017 | 0.003 | 0.012 | 0.003 | 0.008 |
| $Q_2$ | 0.031 | 151 | 0.020 | 104 | 0.012 | 75 |
| $Q_3$ | 0.058 | 606 | 0.022 | 418 | 0.020 | 298 |
| $Q_4$ | 0.091 | 3028 | 0.044 | 2088 | 0.034 | 1492 |
| $Q_5$ | 0.124 | - | 0.072 | - | 0.064 | - |
| $Q_6$ | 0.151 | - | 0. 110 | - | 0.096 | - |
| $Q_7$ | 0.172 | - | 0. 162 | - | 0.122 | - |
| $Q_8$ | 0.204 | - | 0.196 | - | 0.144 | - |
| $Q_9$ | 0.259 | - | 0. 220 | - | 0. 184 | - |

**Figure 20.** A query involving many background joins.

This is indeed the worst-case scenario for both indexes. As shown in Figure 20, the response of the Oracle's Semantic Technology after the 4rd level, was larger than 5000 seconds, thus we terminated the queries. On the other side, although the execution time using our index increases at each level, the important thing is that this increase remains fairly acceptable, for such type of extreme queries. The GS index results to faster background queries because the graph signature fits in a small memory, even with some magnitudes of self joins. Oracle's Semantic Technology, on the other hand, performs the self-joins on the whole dataset, which is too large. In other words, the GS index joins only the Graph Signature, which is 1M edges, whereas 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's size.

## 7.4 Usability Evaluation

To evaluate how easy it is to use MashQL, we invited 40 people to use the MashQL editor to formulate basic and advanced queries over RDF datasets found at http://data.semanticweb.org, which contains over 80k triples about articles, people, organizations, conferences, and workshops. 25 participants were non-IT skilled (i.e., had only basic skills for web browsing); and other 15 were IT-skilled people – but none of them was familiar with RDF or SPARQL. A 10-minutes tutorial about MashQL was given before the evaluation started, illustrating examples of MashQL queries but no hands-on exercises or examples from the datasets used.

Each of the 40 participants was given 6 queries to formulate (listed in Figure 21). After formulating each query in MashQL, each person was asked to manually browse the queried page(s) and compose the answer. The average time needed to formulate each query in MashQL (versus the manual navigation) was recorded and is presented in Figure 22(a). The time needed to formulate a query by the IT-skilled (versus the non IT-skilled) is presented in Figure 22(b).

After finishing all queries, each person was asked to fill in a questionnaire that evaluated the main features of MashQL. The results are summarized in Figure 23.

This evaluation included the MashQL editor and the Firefox add-on. The evaluation conclusions for each case were almost the same, thus they are merged here for the sake of brevity. We refer to [49] for more details about each evaluation.

**Q1.** *"Find the titles of the articles presented at the 4th European Semantic Web Conference"*. This is a simple query and helps to get familiar with the MashQL editor.

**Q2.** *"To learn more about these articles, find the titles, authors, and abstracts of the articles presented at the 4th European Semantic Web Conference"*. This extends Q1, but this time the participants have to compare the easiness of the web navigation with the use of MashQL.

**Q3.** *"Retrieve all the titles, authors, and abstracts of the articles presented at the 4th European Semantic Web Conference that have a title that contains the word Semantic"*. A more difficult query, to show the querying efficiency of the two methods.

**Q4.** *"Update the previous query by retrieving also and the homepages of the authors, and order the results"*. This scenario emphasizes the ordering functionality of the editor compared to the manual ordering of the information, after gathering them in a file.

**Q5.** *"Retrieve the names of all authors of papers that contain the word 'Semantic' and presented in the 4th European Semantic Web Conference"*. In the previous queries one should start with "Article", here it should start with "Person".

**Q6.** *"Retrieve the names and homepages of the authors that attended the 4th European Semantic Web Conference or the 16th International World Wide Web Conference, and the authors' names contain the word Thomas'*. This query is a mashup involving multiple pages/sources, and contains the "OneOf" operator.

**Figure 21.** List of queries used in the evaluation.

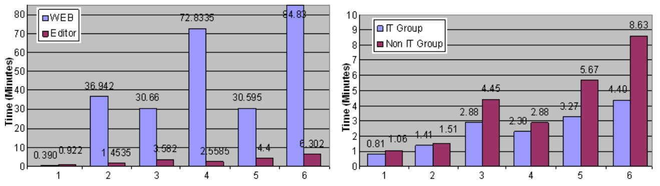


**Figure 22.** (a) Manual Navigation vs. MashQL; (b) IT vs. non IT.

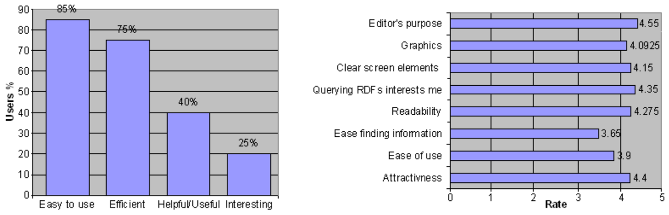


**Figure 23.** Evaluation of the editor interface and features.

We found that most of the people were generally happy and the core ideas of MashQL were appreciated. People were able to learn MashQL quickly by practicing it; if they were to perform similar queries on other datasets they would do it much faster next time. It is worth noting that none of the 40 people failed to formulate the given queries. We also observed that people are still not used with the Data Web paradigm (i.e., dealing with structured data and the difficulty of querying it). They are used to "google" information and then manually navigate to compose answers, without noticing how much time they consume or the impreciseness of the results.

Since MashQL is not intended to be used by developers (e.g., SPARQL and RDF experts), but rather, by people who are unfamiliar with these technologies, our usability study did not compare MashQL usability with SPARQL usability. However, it is worth noting that some users of the MashQL editors have used it to learn SPARQL. The tool supports a debugging functionality that displays the generated SPARQL script, and allows one to directly change this script, and then look back to these changes in MashQL. This indicates that the MashQL's intuition is easier to learn for SPARQL beginners.

## 8. Conclusions and Future Work

We proposed a query formulation language, called MashQL. We have specified four assumptions that a Data Web query language should have, and shown how MashQL implements all of them. The language-design and the performance complexities of MashQL are fundamentally tackled. We have designed and formally specified the syntax and the semantics of MashQL, as a language, not merely a single-purpose interface. We have also specified the query formulation algorithm, by which the complexity of understanding a data source (even it is schema-free) are moved to the query editor. We addressed the challenge of achieving interactive performance during query formulation by introducing a new approach for indexing RDF data. We presented two different implementation scenarios of MashQL and evaluated our implementation on two large datasets.

We plan to extend this work in several directions. We will introduce a search-box on top of MashQL to allow keyword-search and then use MashQL to filter the retrieved results. To allow people use MashQL in a typical data integration scenario, several reasoning services will be supported, including SameAs, Subtype, Sub-property, and Part-of. Furthermore, we are collaborating with colleagues to use MashQL as a business rules language, thus include several reaction and production operators. We plan to also support aggregation functions, as soon as their semantics are defined and standardized in SPARQL. Supporting such functions in MashQL is not difficult since we only need to allow the user to select a function (e.g., sum, avg, max, etc.) before a subject, property or object. Last but not least, we are currently extending the Graph Signature approach for general-purpose query optimization. In particular, we are seeking to extend the Graph Signature to optimize arbitrary SPARQL queries; for this, we need to extend our query model to retrieve not only the last node/edge, but any node/edge, as well as star-shaped queries. This is not difficult, because a query path is the building block for star-shaped queries. Furthermore, we plan to use our approach on keyword-search. In such a scenario, we expect to have fast responses, because false positives are less important. Last but not least, we need to develop a maintenance strategy to support querying dynamic datasets.

## Acknowledgements

This research was partially supported by the SEARCHiN project (FP6-042467, Marie Curie Actions). We would like to also thank Ala Hawash, Andreas Manoli, Anton Deik, Bilal Farraj, and Constantinos Savvides for their support and help in the implementation of MashQL.

## References

1 Altova XMLSpy®: http://www.altova.com/solutions/xquery-tools.html (Feb. 2010)
2 Stylus Studio®: http://www.stylusstudio.com/xquery_editor.html (Feb. 2010)
3 Isparql: http://lod.openlinksw.com/isparql (Feb. 2010)
4 RDB2RDF: http://www.w3.org/2005/Incubator/rdb2rdf (Feb. 2010)
5 SPARQL Extensions: http://esw.w3.org/SPARQL/Extensions? (Feb. 2010)
6 SparqlMotion: http://www.topquadrant.com/sparqlmotion (Feb. 2010)
7 Yahoo Pipes: http://pipes.yahoo.com/pipes (Feb. 2010)
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. ISWC2004.
10 BEA Systems, Inc.: BEA AquaLogic Data Services Platform™ - XQuery Developer's Guide. Version 2.5, 2005.
11 Bloesch A, Halpin, T: Conceptual Queries using ConQuer–II. ER 1997.
12 Comai S, Damiani E: Computing Graphical Queries over XML Data. ACM Transactions on Information Systems, 19(4). 2001
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 De Keukelaere F, Bhola S, Steiner M, Chari S, Yoshihama S:SMash: secure component model for cross-domain mashups on unmodified browsers. WWW 2008.
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 Dionisiof J, Cardenasf A: MQuery: A Visual Query Language for Multimedia, Timeline and Simulation Data. J. Visual Languages & Computing, 7(4).1996
18 Goldman R, Widom J: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. VLDB 1997.
19 Griffin E: Foundations of Popfly. Springer, 2008.
20 Henzinger R, Henzinger A, Kopke W: Computing Simulations on Finite and Infinite Graphs. FOCS 1995.
21 Hofstede A, Proper H, and Weide T: Computer Supported Query Formulation in an Evolving Context. Australasian DB Conf. 1995.
22 Jarrar M: Towards Methodological Principles for Ontology Engineering. PhD Thesis.Vrije Universiteit Brussel. 2005
23 Jarrar M, Dikaiakos M: A Data Mashup Language for the Data Web. Proceedings of LDOW, at WWW'09. ISSN 1613-0073. 2009.
24 Jarrar M, and Dikaiakos M: MashQL: Querying the Data Web. IEEE internet computing. April 2010.
25 Jarrar M, Dikaiakos M, Querying the Data Web, tech. report TR-09-04, Dept. Computer Science, Univ. of Cyprus, Nov. 2009; http://encs.birzeit.edu/klab/publications/TARMD10.pdf.htm
26 Jagadish H, Chapman A, Elkiss A, Jayapandian M, Li Y, Nandi A, YuC: Making database systems usable. SIGMOD 2007
27 Jarrar M, Dikaiakos M.: Technical Article: Querying the Data Web. University of Cyprus. www.cs.ucy.ac.cy/~mjarrar/TAR200904.pdf
28 Jayapandian M, Jagadish H:Automated Creation of a Forms-based Database Query Interface. VLDB 2008.
29 Jayapandian M, Jagadish H:Expressive Query Specification through Form Customization. EDBT 2008.
30 Kaufmann E, Bernstein A: How Useful Are Natural Language Interfaces to the Semantic Web for Casual End-Users. ISWC, 2007.
31 Kaushik R, Bohannon P, Naughton J, Korth H: Covering Indexes for Branching Path Queries. SIGMOD 2002.
32 Kaushik R, Shenoy P, Bohannon P, Gudes E: Exploiting local similarity for indexing of paths in graph structured data. ICDE 2002.
33 Li Y, Yang H, Jagadish H: NaLIX: an interactive natural language interface for querying XML. SIGMOD 2005.
34 Manoli A: MashQL Implementation in the Firefox Browser. B.Sc. Thesis. Computer Science dept., University of Cyprus, Dec. 2009.
35 Miller R: Response time in man-computer conversational transactions. AFIPS 1968
36 Milo T, Suciu D: Index structures for path expressions. ICDT 1999.
37 Nandi A, Jagadish H: Assisted querying using instant-response interfaces. SIGMOD 2007.
38 Nestorov S, Ullman J, Wiener J, Chawathe S: Concise Representations of Semistructured Hierarchical Data. ICDE 1997.
39 Neumann T, Weikum G: RDF3X: RISC style engine for RDF. VLDB'08
40 Paige R, Tarjan R: Three partition refinement algorithms. SIAM Journal on Computing, 16(6):973–989. 1987.
41 Parent C, Spaccapietra S: About Complex Entities, Complex Objects and Object-Oriented Data Models. Information System Concepts, 1989
42 Petropoulos, M, Papakonstantinouy, Vassalos V: Graphical Query Interfaces for Semistructured Data ACM Internet Technology, 5(2). 2005
43 Prud'hommeaux E, Seaborne A: SPARQL Query Language for RDF. 2008
44 Popescu A, Etzioni O, Kautz H: Towards a theory of natural language interfaces to databases. 8th Con on Intelligent user interfaces. 2003
45 Russell A, Smart R, Braines D, Shadbolt R.: NITELIGHT: A Graphical Tool for Semantic Query Construction. SWUI Workshop. 2008.
46 Steer D, Miller L, Brickley D: RDFAuthor: Enabling everyone to author rdf," WWW'02 Developers Day, 2002.
47 Stockmeyer L, Meyer A: Word problems requiring exponential time. STOC'73.
48 Tummarello G, Polleres A, Morbidoni C: Who the FOAF knows Alice? ISWC Workshops. 2007
49 Savvides C: MashQL: A Step towards Semantic Pipes. M.Sc. Thesis. Computer Science dept., University of Cyprus, May 2010.
50 Zloof M: Query-by-Example: Data Base Language. IBM Systems 16(4). 1977

## APPENDIX

**Proof of Theorem 1.** Given a query $Q$ ($o_n$:{$o_1\ p_1\ o_2\ p_2 \ldots p_{n-1}\ o_{n-1} p_n\ o_n$}). Evaluating this query safely means the results set $\boldsymbol{o}_x$ when evaluating $Q$ on a data graph $G$, is a subset of the results $O_x$ of evaluating it on the O-Signature ($\boldsymbol{o}_x \in \boldsymbol{O}_x$).. Let $p_1$ be an edge from $o_1$ to $o_2$ in $G$, then (by definition), its O-Signature $S_O$ must contain $p_1$ from $O_1$ to $O_2$, and for $p_n$ from $o_{n-1}$ to $o_n$ there is $p_n$ from $O_{n-1}$ to $O_n$ (where $o_1 \in O_1 \ldots o_n \in O_n$). Thus for every path ($o_1 \xrightarrow{p1} o_2 \xrightarrow{p2} \ldots\ o_{n-1} \xrightarrow{pn}$) in $G$ there is exactly the path ($O_1 \xrightarrow{p1} O_2 \xrightarrow{p2} \ldots\ O_{n-1} \xrightarrow{pn}$) in $S_O$. When evaluating $Q$ on $G$ we retrieve $\boldsymbol{o}_x$ the set of nodes having this path into them, and similarly $\boldsymbol{O}_x$ when evaluating $Q$ on $S_O$. Since each $o_i \in O_i$, then ($\boldsymbol{o}_x \in \boldsymbol{O}_x$).

**Proof of Theorem 2.** Given a query $Q$ ($p_n$:{$o_1\ p_1\ o_2\ p_2 \ldots p_{n-1}\ o_{n-1}\ p_n\ o_n$}). Evaluating this query precisely means that the results set $\boldsymbol{p}_x$ when evaluating $Q$ on $G$, is exactly the same results $\boldsymbol{P}_x$ when evaluating it on the $S$o ($\boldsymbol{p}_x = \boldsymbol{P}_x$). Now, let $p_1$ be an edge from $o_1$ to $o_2$ in $G$, then (by definition) its O-Signature $S_O$ must contain $p_1$ from $O_1$ to $O_2$, and so for $p_n$ from $o_{n-1}$ to $o_n$ there is $p_n$ from $O_{n-1}$ to $O_n$ (where $o_1 \in O_1 \ldots o_n \in O_n$). Hence for every path ($o_1 \xrightarrow{p_1} o_2 \xrightarrow{p_2} \ldots\ o_{n-1} \xrightarrow{p_n} o_n$) in $G$ there is exactly the path in $S_O$ ($O_1 \xrightarrow{p_1} O_2 \xrightarrow{p_2} \ldots\ O_{n-1} \xrightarrow{p_n} O_n$); and since the path ($\xrightarrow{p_1} \xrightarrow{p_2} \ldots \xrightarrow{p_n}$) from $o_1$ to $o_{n-1}$, is (by definition) the same path from $O_1$ to $O_{n-1}$. Then, the set of edges $\boldsymbol{p}_x$ from $o_{n-1}$ to $o_n$ is exactly the same set of edges $\boldsymbol{P}_x$ from $O_1$ to $O_n$.

**Proof of Theorem 3.** Given query $Q$ ($o_n$:{$o_1\ p_1\ o_2\ p_2 \ldots p_{n-1}\ o_{n-1}\ p_n\ o_n$}) where every node $o$ is a variable. Evaluating this query precisely means retrieving all nodes $\boldsymbol{o}_x$ having the path ($\xrightarrow{p1} \xrightarrow{p2} \ldots \xrightarrow{pn}$) into them, i.e. regardless of which previous nodes this path follows to reach them, as all previous nodes are variables. Hence, by definition, for every node in $\boldsymbol{o}_x$ having the path ($\xrightarrow{p1} \xrightarrow{p2} \ldots \xrightarrow{pn}$) into it in $G$, there is the same path ($\xrightarrow{p1} \xrightarrow{p2} \ldots \xrightarrow{pn}$) into $\boldsymbol{O}_x$ in $S_I$. Since, also by definition, the nodes in $\boldsymbol{O}_x$ are exactly the nodes in $\boldsymbol{o}_x$ that have the same paths, then $\boldsymbol{o}_x = \boldsymbol{O}_x$.

**Proof of Theorem 4.** Because (as proven in theorem 1) the answer of the O/I-Signature is safe, i.e., a superset of the target answer, then if this superset is empty the target answer is empty.

## About the Authors

**Mustafa Jarrar** (mjarrar@birzeit.edu) is an assistant professor at the University of Birzeit in Palestine. His research interests include the Semantic Web, ontology engineering, databases, Web 2.0, and data mashups. Jarrar has a PhD in computer science from Vrije Universiteit Brussel in Belgium; and an Experienced Marie Curie fellow at the University of Cyprus. He is a full member of the IFIP2.6 on Database Semantics, the IFIP2.12 on Web Semantics, and the IEEE Learning Standards Committee.www.jarrar.info

**Marios D. Dikaiakos** (mdd@cs.ucy.ac.cy) is an associate professor of computer science at the University of Cyprus. His research interests include network-centric computing systems and Web technologies. Dikaiakos has a PhD in computer science from Princeton University. He's a senior member of the ACM and a member of the IEEE Computer Society and the Technical Chamber of Greece. www.cs.ucy.ac.cy/~mdd