Querying the Data Web

The MashQL approach

Mustafa Jarrar
Birzeit University, Palestine

Marios D. Dikaiakos
University of Cyprus, Cyrus

Abstract. *We propose a query formulation language, called MashQL for querying and mashing up structured data on the web. The novelty of MashQL is that it does not require one to know the structure of the queried data or the data itself to adhere to a schema. 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 this article. A novel technique to optimize queries over large graph dataset is proposed in order to allow instant user-interaction. The user-interaction is successfully evaluated on DBPedia, which is a large and noisy dataset.*

In parallel to the continuous development of the hypertext Web, we are witnessing an explosion of structured data published on the Web and leading to the emergence of a "Web of Data." Companies like Google, Freebase, Upcoming, eBay, Yahoo, and Amazon, are competing on gathering and publishing structured content, and on encouraging people to reuse and profit from it. To facilitate their push for content reuse, these companies have started to adopt advanced Web metadata standards and promote the use thereof. For example, Yahoo supports Web-sites pointing to or embedding RDF by better presenting them in the search results; MySpace adopted RDF for profile and data portability; Upcoming publishes its content in microformats and will support RDFa, a W3C recommendation for the embedding of RDF inside XHTML.

The trend of publishing structured data on the Web is shifting the focus of Web technologies towards new paradigms of *structured-data retrieval*. Traditional search engines cannot serve such data adequately because a keyword-based query will still be ambiguous even though 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 and fuse this data easily.

Unlike keyword-based information retrieval, the main challenge in structured-data retrieval 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 should not be expected to investigate "what is the schema" when seeking information. In many cases, however, a schema can be dynamic, with many kinds of items with different attributes being added and dropped from a dataset. Other data sources might be schema-free (i.e., without a stated schema to describe its structure) or the schema might consist of assertions mixed up with the data (e.g., RDF). In RDF, it is not necessary for the data to be consistent with (i.e., adhere to) a schema or ontology; therefore, one has to manually investigate the RDF data itself before querying it, in order to understand the vocabulary and the data structures. This becomes more challenging when a query involves multiple RDF sources. As discussed in related work, allowing end-users to easily search and consume data is a general and a known challenge in different areas. For a query language to be practically sound in the context of an open environment, as the Web of Data, the following assumptions should be held:

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

This article presents a query formulation language, called *MashQL*. The novelty of MashQL is that it considers all of the assumptions stated above. 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. We only assume an RDF source to be syntactically correct.

MashQL is a general-purpose query formulation language; however, this article focuses on the Data Web scenario. We regard the Web as a database, where each data source is regarded as a "table". In this view, a *data mashup* can be regarded as a query involving multiple data sources. To illustrate the power of MashQL, we chose to focus on RDF, not only as it is the most challenging, but because it is the most primitive data model, i.e., other models (e.g., XML and databases) can be easily mapped into it.

MashQL

Figure 1 shows two RDF sources and a MashQL query to retrieve "anything written by Lara, published after 2007, and with a title". The first module specifies the input and the second specifies the query. The output can be piped into a third module (not shown here), which renders the results into a certain format (e.g., HTML or XML), or as RDF input to other queries. As it will be explained in the query formulation section, such a query is formulated interactively, without any prior knowledge of the schema and without assuming that

Related Work

Query-By-Form is the simplest approach for queries, but not flexible.

Query-By-Example. This approach requires the data be schematized and the users to be aware of the schema.

Conceptual Queries. As many databases have EER or ORM conceptual diagrams describing them, one can query a database starting from its diagrams [1]. However, understanding conceptual diagrams is still difficult for non-experts.

Natural Language Queries allow people to write their queries as natural language sentences, and translate them into e.g., SQL [2], or XQuery [3]. The problem is that these approaches are fundamentally bounded with the language ambiguity.

Mashup Editors and Visual Scripting. Some mashup editors (e.g., Yahoo Pipes, Popfly, sMash) allow people to write query scripts inside a module, and visualize these modules 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). **Deri Pipes** [4] are inspired by *visual scripting*. They allow users to write SPARQL queries, in a *textual form*, inside a box and link it to other boxes. Deri Pipes focus on the pipelining aspects, such as what operators are needed in a pipeline. MashQL is also inspired by the way Yahoo visualizes query modules. However, and in contrast to Deri/Yahoo Pipes, its main purpose is query formulation itself, i.e., what is inside a query model. Hence, MashQL is a complementary, rather than an alternative, to Yahoo and Deri Pipes.

Interactive Queries. The closest approach to MashQL is Lorel [5], which was developed for querying scheme-free XML. The difference: i) Lorel partially handles schema-free queries. Instead of querying the original data, it queries a summary of the data (called *dataguide*). A DataGuide contains only possible paths between predicates, i.e., it groups unrelated items having same property labels. ii) Lorel does not support querying multiple sources; and iii) its expressivity is very basic. However, MashQL supports path disjunctions, negations, variables, union, reverse properties, among others.

References

1. Bloesch A, Halpin, T: Conceptual Queries using ConQuer–II. ER'97.
2. Li Y, Yang H, Jagadish H: NaLIX: an interactive natural language interface for querying XML. SIGMOD'05.
3. Popescu A, Etzioni O, Kautz H: Towards a theory of natural language interfaces to databases. IUIC'03.
4. Le Phuoc D, Polleres A, Morbidoni C, Hauswirth M, Tummarello G: Rapid semantic web mashup development through semantic web pipes. WWW'09.
5. Goldman R, Widom J: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. VLDB'97.

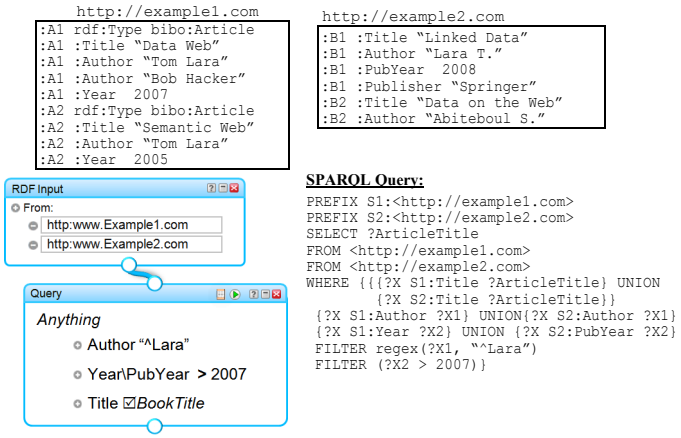


Figure 1. A query over two RDF data sources.

data adheres to a schema or ontology. The union operator "\" allows the combination of two properties within or across data sources (e.g., Year\PubYear). Notice that although MashQL can be used for data integration and fusion, this is not a goal per se for MashQL. Data integration requires not only syntax, but also *semantic* integration, which MashQL does not support. MashQL is mainly designed for query formulation, by which people are able to spot different labels of same properties (as they navigate through datasets) and to manually combine them, as the previous example illustrates.

The Intuition of MashQL

Each MashQL query is seen as a tree. The root is called the *query subject,* which is the subject matter being inquired. A subject can be a particular instance, an instance type, or a user-defined variable label. Each branch of the tree is called a *restriction* and is used to restrict a certain property of the subject. Branches can be expanded to allow sub-trees, called *query paths*. In this case, the object of a property is seen as the subject of its sub-query. In this way, one can navigate through the underlying dataset and build complex queries. Objects marked with "☑" will be returned in the query results, i.e., projection. When querying different sources, two properties (or two instances) are considered the same if they have the same URI.

Example. To illustrate the notion of query paths, in Figure 2 we show a query to retrieve the recent articles from Malta, i.e., the title of every article that has an author, this author has an address, this address has a country "Malta", and the article is published after 2007.

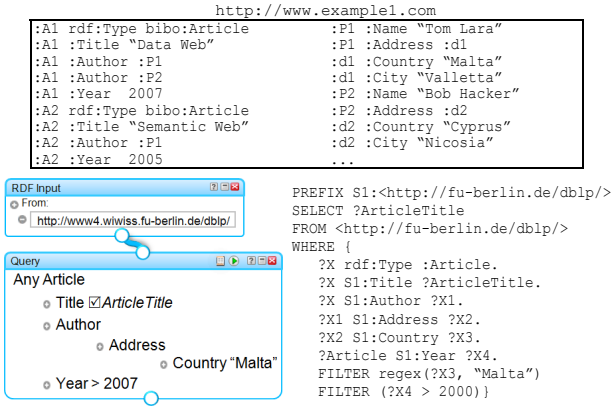


Figure 2. Query paths (/sub trees) in MashQL

Query Formulation

Formulating a query is an interactive process, during which a user performs selections from drop-down lists. While the user interacts with the query editor, the editor queries the dataset in the background to generate these lists. After specifying the input sources (i.e., dataset), the user first selects from the **subjects list**, which contains either: (i) the set of the subject-types, e.g., Article; or (ii) the union of all subject and object identifiers in the dataset, such as A1 or B2. The user can also choose to (iii) introduce a new *subject label*, which is then considered a variable and displayed in *italics*. The default variable is *Anything*. The user then selects from the **properties list**, which is generated on the fly and comprises all properties pertinent to the chosen subject. *Finally,* the user selects a **restriction**: if she wants to add an object filter on the previously selected property, a list will be offered {Equals, Contains, etc.}. If the user wants to add an object identifier as a restriction, a list of the possible objects will be generated (depending on the previous selections). The user can also choose to expand the property to declare a query path. In this way, users can navigate and query a data graph, without prior knowledge of it, even if the data is schema-free.

The formal algorithm to generate the drop-down lists can be found in section 4 in the extended version of the article [1]. Issues related to the achievement of acceptable interactive performance in the presence of large datasets, the normalization of cryptic URIs, and the scalability of drop-down lists are discussed in the remaining sections.

Syntax and Semantics

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 [2]. Thus, when evaluating a query $Q(S)$, only the triples satisfying all restrictions are retrieved, such that: i) If a restriction is not prefixed with any modal operator (R:=<*empty, P, O_f*>), its evaluation is considered true if the subject, the predicate, and the object-filter are matched (see Def. 3 in Table I). This case is mapped into a normal triple pattern in SPARQL. ii) If a restriction modality is "maybe" (R:=<*maybe, P, O_f*>), its evaluation is always true, and mapped into an optional triple pattern in SPARQL. iii) If a restriction is modality "without" (R:=<*without, P, O_f*>), its evaluation is true if the subject S and the predicate P do not appear together in a triple, i.e., the object O should not be bound. Table I presents a summary of the formal definitions of MashQL; the full SPARQL interpretation can be found in [1].

Example. When evaluating the query in Figure 3, we retrieve all RDF triples with the same subject that have: 1) a predicate Title, 2) a predicate Artist with an object identifier being Shakera, 3) an optional predicate Album, and 4) do not have the predicate Copyright.

MashQL supports 9 forms of filters and query paths (Def. 4) ; four forms of unions (see Def.5); formulating queries at the type level (Def. 6); exploring an RDF graph backward(Def.7), not only forward.

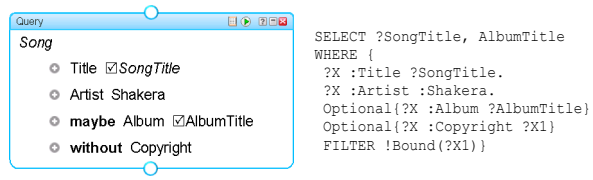


Figure 3. A MashQL query and its mapping into SPARQL.

Table I: The formal definition of MashQL

Def.1 (Query): *A Query Q with a subject S, Q(S), is a set of conjunctive restrictions on S.* $Q(S) := R_1 \wedge \ldots \wedge R_n$.

Def.2 (Subject): *A subject* $S \in I \cup V$, *I is an identifier, V is a variable.*

Def.3 (Restriction): *A restriction* $R := \langle R_x, P, O_f \rangle$, R_x *is a modal operator,* $R_x \in \{empty, maybe, without\}$; *P is a predicate;* $P \in I \cup V$; O_f *is an object filter.*

Def.4 (Object Filter): *An object filter* $O_f := \langle O, f \rangle$, *O is an object, f is a filtering function. f can have one of the following nine forms:*

1. $O_f := \langle O \rangle$, *where O is an object,* $O \in V \cup I$.
2. $O_f := \langle O, Equals(X, T, L_t) \rangle$, *where X can be a variable or a constant, T a datatype, and* L_t *a language tag.*
3. $O_f := \langle O, Contains(X, T, L_t) \rangle$.
4. $O_f := \langle O, MoreThan(X, T) \rangle$.
5. $O_f := \langle O, LessThan(X, T) \rangle$.
6. $O_f := \langle O, Between(X, Y, T) \rangle$.
7. $O_f := \langle O, OneOf(V) \rangle$, *where V is a set of values* $\{v_1 \ldots v_n\}$.
8. $O_f := \langle O, Not(f) \rangle$, *where f is one of the functions defined above.*
9. $O_f := \langle O, Q_i(O) \rangle$, *where O is an object* $O \in V \cup I$, *and* $Q_i(O)$ *is a sub-query with O being the query subject.*

Def.5 (Union): A union operator can be defined as the following :

1. $O_n = \langle O_1 \backslash O_2 \backslash \ldots \backslash O_n \rangle$, *unions between objects,* $O_i \in I$.
2. $P_n = \langle P_1 \backslash P_2 \backslash \ldots \backslash P_n \rangle$, *unions between predicates,* $P_i \in I$.
3. $S_n = \langle S_1 \backslash S_2 \backslash \ldots \backslash S_n \rangle$, *unions between subjects, where* $S_i \in I$.
4. $Q_n = \langle Q_1 \backslash Q_2 \backslash \ldots \backslash Q_n \rangle$, *unions between queries,*

Def.6 (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.7 (Reverse): $\langle \sim P \rangle$ *the reverse of the predicate P. Let* R_1 *be a restriction on S such that* $\langle S\ P\ O \rangle$, *and* R_2 *be* $\langle O \sim P\ S \rangle$, *then* $R_1 = R_2$.

To conclude, MashQL is not merely a single-purpose interface, but rather, a general query formulation language, with all of the assumptions introduced earlier in mind. It is as expressive as SPARQL.

The MashQL Editor

We implemented MashQL in two scenarios: a server-side query and mashup editor, and a browser-side Firefox plug-in. The former is illustrated in Figure 4. Its functionality includes: a state-machine dispatching the "background queries"; translating a formulated MashQL query into SPARQL; executing a query and rendering its results. MashQL queries can be materialized and published if needed. The output of each published query is given a URL,

and seen as a concrete RDF source, i.e., materialized and stored physically. Cyclic queries are not allowed, i.e., the input of a query cannot be its output, directly or through a chain. See the formalization, caching, materialization, and refreshing schemes in section 6.2 in [1].

When a user specifies a data source as input, it is bulk-loaded into Oracle 11g. Subsequently, the MashQL Editor uses AJAX to dispatch background queries and the SPARQL translation for execution by the Oracle 11g. We chose Oracle 11g for its native RDF support and for DBMS functionalities such as materialization, indexing, and partitioning.

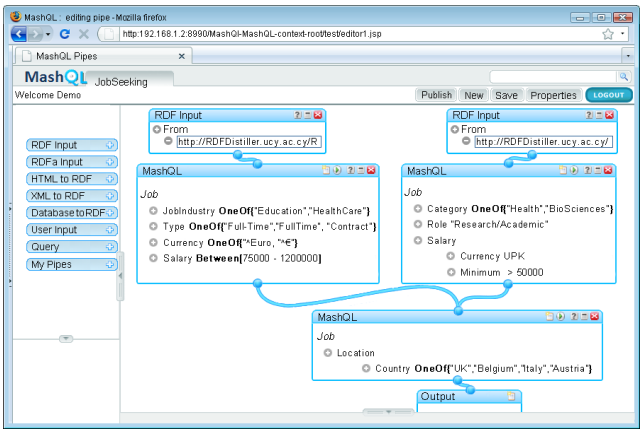


Figure 4. Screenshot of the server-side MashQL-Editor

For the GUI of MashQL, we adopted the Yahoo Pipes style for visualizing Web feed mashups, and used Yahoo's open-source Javascript libraries. Hence, a data mashup becomes a query over multiple data sources, and its setup follows the simple paradigm of Web feed filters and mashups. It is worth noting that the examples of this article cannot be built using Yahoo pipes: Yahoo allows a *limited* support of XML mashups, but this is neither graphical nor intuitive; as one has to write complex scripts in YQL, the Yahoo Pipes' query language.

Our second implementation (Firefox plug-in) has the same functionality as the online editor. However, no databases are used in the back-end. Queries are executed inside the browser, using the Jena SPARQL query libraries. The goal of this implementation is to allow querying and fusing Web-sites that embed RDFa [1,7]. In this way, the browser is used as a Web composer rather than only a navigator.

Implementation Issues

URI Normalization: As RDF may contain unwieldy URIs, queries might be inelegant. Thus, the editor normalizes URIs and *displays* the normalization instead; e.g., Type instead of www.w3.org/1999/02/22-rdf-syntax-ns#type. If one moves over 'type', its URI is displayed as a 'tip'. Different URIs leading to the same normalization are distinguished with a gray prefix (1:type, 2:type). The normalization is based on a repository we built for the common namespace prefixes (e.g., rdf, owl, foaf), which can be also edited by the user. The editor uses heuristics for other cases; e.g., it takes the last part of a URI, after '#'. If '#' doesn't exist, the part after '/'. The result should be at least 3 characters and start with a letter; otherwise it takes the last two parts of the URI, and so on. Our experiments, on many datasets, show that this way covers the majority of cases. However, there is no guarantee to always produce elegant normalization.

Verbalization: To further improve the elegance of MashQL, we implemented a *verbalization* mode. After editing a restriction and moving to edit another, all control boxes and lists in this restriction are verbalized, (their content is converted into friendly text) and display the verbalization instead. If the user comes back to edit a restriction, by clicking on it, then it switches to the edit mode (control boxes and lists are made visible again) and all other restrictions switch to the verbalization mode. That is, only one restriction will have the edit mode at a time. This makes the query representation closer to natural language and facilitates query validation by the users.

Scalable lists: When querying large datasets, the usual drop-down list becomes not 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 point to them. Our list supports also scalable scrolling. The first 50 results are displayed first, but one can scroll to go to the next, arbitrarily middle, or last 50. Each time the editor sends an AJAX query to fetch only those 50. Furthermore, our list allows the user to select either Instances (i.e., any URI in the dataset) or types of instances (i.e., instances that have the predicate rdf:type if exits, such as Author, Person, University),

Performance Considerations

When formulating a MashQL query, the editor queries the dataset in the background to generate the list of next choices. In such a user-interaction setting, the response-time should be small, preferably less than 100ms [3]. As our evaluations in [1] show, such a short response-time cannot be achieved by existing SQL-backed RDF querying technologies [4,5] over large graphs that cannot fit in memory. This is because querying a graph stored in a relational table (S,P,O) involves many self-joins, which are expensive even if this table is well indexed. For large data stores, we suggest constructing the *Graph-Signature*, a graph index designed to enhance the interactivity of the MashQL editor. The idea (similar to XML summaries [6]) is to generate a small-size summary of an RDF graph, so that the editor's background queries can be answered from

this summary, faster than querying the original graph. The Graph Signature groups into the same *category* all RDF nodes with the same set of *outgoing* paths. A category is the set of all subjects that: i) have exactly the same property labels, and ii) the objects of each of their properties belong to same categories. Formal definitions follow:

Def.8: Two RDF subjects S_1 and S_2 have the same **Category** C_i, if and only if:

1. *They share the same property labels*: There exist (S_1 P_1 $O_{1,1}$) … (S_1 P_m $O_{1,m}$), m>0, and (S_2 P'_1 $O_{2,1}$) … (S_2 P'_n $O_{2,n}$), n>0, such that the set $\{P_1,...,P_m\}$ of properties of S_1 is equal to the set of properties of $S_2\{P'_1,..,P'_n\}$.
2. *The objects of each property of S_1 and S_2 belong to the same category*: For each property P_i of S_1 and S_2, its corresponding objects $O_{1,i}$ and $O_{2,i}$ belong to a same category C_j.

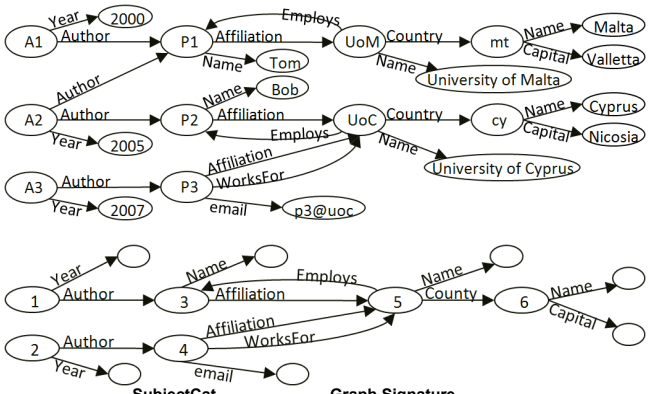
Def.9: A **Category signature** is a set of triples of the form <SC, P, OC>, where SC is a category of some subjects, P is a property label, and OC is a category of some objects.

Def.10 Graph Signature is the set of all category signatures. A Graph Signature is also a directed labeled graph as RDF.

Example. Figure 5 shows an RDF graph and its graph signature, represented also in a table format. The SubjectCat table indexes the extent of the categories. Notice that P1 and P2 are assigned the same category 3, as they share the same set of property labels {Affiliation, Name}. P3 is assigned another category 4, as its properties are not the same as P1 and P2. A1, A2 and A3 have the same properties but A3 is assigned a different category, because (see Def.8.2) the object of its author has another category. That is, the category of A1.Author and A2.Author is 3, while the category of A3.Author is 4.

Querying the Graph Signature is similar to querying an RDF graph. Our query model is much simpler than SPARQL, which is $O_n|P_n:\{O_1\ P_1\ O_2\ P_2\ .\ .\ .\ P_n\ O_n\}$, where O_i is a node, P_i is a predicate and each can be a constant or a variable. Notice we only retrieve/project the last node or predicate label ($O_n|P_n$), which is sufficient for our query formulation purposes. Evaluating such a query is illustrated with the following example: Suppose we need to know "What are the properties of the countries of the affiliations of the authors of A3." To answer this query we have to first look up the category of A3, which is 2, and then generate the SPARQL query: SELECT P WHERE {:2 :Author ?O1)(?O1 :Affiliation ?O2)(?O2 :Country ?O3)(?O3 ?P ?O5)}. Answering such queries from the Graph Signature is fast because its size typically is small.

Given an RDF graph with size *n*, where *n* is the number of the triples, this graph can be summarized by *m* triples, where $1 \leq m \leq n$. If a graph is heterogeneous, i.e., there are



SubjectCat

S	C
A1	1
A2	1
A3	2
P1	3
P2	3
P3	4
UoM	5
UoC	5
mt	6
cy	6

Graph Signature

SC	Predicate	OC
1	Author	3
1	Year	-
2	Author	4
2	Year	-
3	Name	-
3	Affiliation	5
4	email	-
4	Affiliation	5
4	WorksFor	5
5	Country	6
5	Name	-
6	Name	-
6	Capital	-

Figure 5. Graph Signature index.

no similar subjects (Def.8), the size of its summary is the same as its original size . In practice, RDF graphs tend to be structurally homogenous, e.g., the DBPedia is summarized by 165K triples. Some annotation triples that are irrelevant (e.g., Label, Comments) can be excluded, and the triples that indicate equivalence (e.g., SameAs, Redirect) can be normalized before computing the Graph Signature. Please refer to the full report [1] for more details. This report also explains that the Graph Signature follows an opposite approach to XML summaries, which is based on incoming –rather than outgoing- paths".

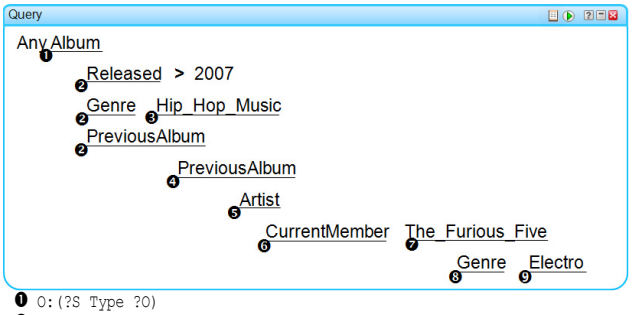
Evaluation

Our evaluation is based on DBPedia, the RDF version of the Wikipedia, which includes 32 million RDF triples (6.7 GB). From this, we extracted four sub-graphs: B32 contains 32M triples; B16 contains 16M triples from B32, B8 contains 8M, and B4 contains 4M (Table II). No sorting is used before partitioning (e.g., `Create B16 As select * from B32 where rownum <16000001`). Each partition is loaded into a table in Oracle 11g, which is installed on a server with 2GHz and 2GB RAM. **Scalability Evaluation:** A Graph Signature is built on each partition. As Table 1 shows, the time cost to build a Graph Signature is linear with the data size; 4M cost 285sec, and 8M cost 528sec. The behavior of the index w.r.t the number of the triples is scalable, e.g., the 32M triples are summarized by 165k triples, and 4M by 56K. Notice that B16 and B8 generated bigger summaries than B32. This is because more similarities are found when all the data is put together.

	Number of	B32	B16	B8	B4
Original Data	Unique Triples	32M	16M	8M	4M
	Unique Subjects	9.4M	6M	4M	2.6M
	Unique Predicates	35	35	34	34
	Unique Objects	16M	8.7M	4.7M	2.5M
	Data Size	6.7GB	3.1GB	1.4GB	550MB
Graph Signatures	Indexing Time (Seconds)	2378	1177	528	285
	Unique Categories	6K	32K	16K	6K
	Triples in Graph Signature	165K	486K	185K	56K

Table II: Statistics about DBPedia and the Graph Signatures.

Response-Time Evaluation: We are not interested in evaluating the execution of the final query; but rather, the queries MashQL performs in the background to generate the "next" drop-down list, i.e., the response-time of the MashQL user-interaction. To formulate the query in Figure 6 over DBPedia, the user first selects the subject from a list. The query produces this list is annotated by ❶. The user then selects a property of this subject from a list, produced by query ❷; and so on. These queries are executed on each partition of the dataset. The cost is shown below.



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

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

Figure 6. Experimental queries and the cost in milliseconds.

As this experiment shows, the cost for all queries remains within few milliseconds regardless of the data size and the length of the join-path expressions. This is because the size of the Graph Signature is small, compared to the original graph. This speed enables the MashQL editor to perform its background queries instantly. More datasets, experiments, and comparisons with Oracle's Semantic Technology can be found in [1].

Conclusions and Future Work

We have presented a query formulation language for the Data Web, called MashQL. MashQL is not merely a single-purpose query interface, but rather, a general and expressive language. MashQL allows people to query multiple sources and without any prior knowledge about their schemes. To achieve instant user-interaction, a novel and effective way is proposed for summarizing large graphs.

Further extensions to MashQL will also support keyword-based queries, reasoning, and aggregation functions. Last but not least, the Graph Signature index will be extended for SPARQL query optimization purposes.

Acknowledgements: This research was partially supported by the SEARCHiN project (FP6-042467, Marie Curie Actions).

References

1 Jarrar M, Dikaiakos M: Querying the Data Web. Technical Report TR-08-04, Department of Computer Science, University of Cyprus, Nov. 2008, http://www.cs.ucy.ac.cy/~mjarrar/TAR200904.pdf

2 Prud'hommeaux E, Seaborne A: SPARQL Query Language for RDF. 2008

3 Miller R: Response time in man-computer conversational transactions.AFIPS'68

4 Chong E, Das S, Eadon G, Srinivasan J: An efficient SQL-based RDF querying scheme. VLDB'05.

5 Neumann T, Weikum G: RDF3X: RISC style engine for RDF. VLDB'08.

6 Nestorov S, Ullman J, Wiener J, Chawathe S: Concise Representations of Semistructured Hierarchical Data. ICDE'97.

7 Jarrar M, Dikaiakos M: A Data Mashup Language for the Data Web. LDOW'09

Keywords: Query Formulation, Semantic Web, Data Web, RDF and SPARQL, Indexing Methods.

Dr. Mustafa Jarrar is an assistant professor at the University of Birzeit in Palestine. He had been appointed as an Experienced Marie Curie Fellow at the University of Cyprus, for two years. Before that he worked as a Senior Research Scientist at STARLab, Vrije Universiteit Brussel in Belgium. He published more than 80 articles and refereed reports in the areas of Semantic Web, ontology engineering, databases Web 2.0, and data mashups. Dr. Jarrar is a full member of the IFIP2.6 on Database Semantics, the IFIP2.12 on Web Semantics, the IEEE Learning Standards Committee, and the CEN/ISSS ICT Skills and Curricula. Dr. Jarrar's h-index is currently 11 and citations +550.

Address:

Birzeit University, P.O. Box 14 Birzeit, Ramallah, Palestine

Telefax: +972 2 2982935

Email: mjarrar@birzeit.edu

Marios D. Dikaiakos is an Associate Professor of Computer Science at the University of Cyprus. Dikaiakos received his Ph.D. from Princeton University (1994), an M.A. degree from Princeton (1991), and a Dipl.-Ing. degree from the National Technical University of Athens (summa cum laude, 1988). His research interests include network-centric computing systems and world-wide web technologies. Dr Dikaiakos is a senior member of the ACM, a member of the IEEE Computer Society and of the Technical Chamber of Greece.

Address:

University of Cyprus, 1678 Nicosia, Cyprus.

Tele: +357-22892700 Fax: +357-22892701

Email: Mdd@cs.ucy.ac.cy